Diversiteit voor softwarebescherming

Diversity for Software Protection

Bertrand Anckaert

Promotor: prof. dr. ir. K. De Bosschere
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2007 - 2008

UNIVERSITEIT
GENT

ISBN 978-90-8578-186-8
NUR 980
Wettelijk depot: D/2008/10.500/5

Aan Elisa, mijn vrouw
en Jasper en Febe, onze kindjes

Voorwoord
–Preface–

Als ik bedenk wat voor een gelukkige samenloop van omstandigheden geleid heeft tot het moment waarop ik dit doctoraat kan afsluiten met het uiten van mijn dankbaarheid, prijs ik me gelukkig dat niemand kan knoeien met het verleden.

In de eerste plaats wil ik mijn ouders bedanken voor de onvoorwaardelijke steun en het vertrouwen dat ze mij geven. Ook mijn broers ben ik bijzonder dankbaar, onder meer voor hun ongebreidelde nieuwsgierigheid naar alles wat ook maar iets met wetenschap en techniek te maken heeft, van het zoeken naar miljoenen jaren oude ammonieten tot bedenkingen bij de futuristische X-files.

Voor de iets minder wetenschappelijke, maar daarom niet minder interessante, vrijetijdsbesteding kon ik altijd terecht bij de Statskaerten. Vinnie, Dorper, Knorry, Tim, en de anderen, ook al zie ik jullie tegenwoordig door alle drukte veel te weinig, merci.

De laatste jaren heb ik het genoegen om naast mijn eigen familie ook nog te kunnen rekenen op die van mijn vrouwtje. Elisa, ge zijt een schatje en ik kijk er naar uit om nog vele jaren samen voor onze kindjes Jasper en Febe te zorgen. We gaan van het leven genieten tot we samen oud en versleten zijn.

Met zoveel steun op persoonlijk vlak durfde ik mij wagen aan de studies Informatica. Van bij het begin kon ik rekenen op Frederik en Werner. Ze zagen het door de vingers dat ik nog nooit van arrays had gehoord en dat een string voor mij nog een heel andere betekenis had. Hun mateloze inzet, in het bijzonder die van Frederik tijdens onze gezamenlijke scriptie, werkte aanstekelijk.

Toen kwam de keuze tussen doctoreren en een “echte” job. Als ik bedenk wat ik allemaal had moeten missen als ik niet voor het doctoraat gekozen had... Ik ben mijn promotor, prof. Koen De Bosschere, dan ook bijzonder erkentelijk voor de begeleiding en de vele kansen die hij mij geboden heeft. Mede dankzij hem kon ik mij gedurende iets meer dan vier jaar verdiepen

in één van de meest uitdagende onderwerpen in de informatica. Het instituut voor de Aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT) stond, samen met Universiteit Gent, in voor de financiële ondersteuning.

Ik wil ook mijn collega's Bjorn, Bruno, Dominique, Ludo en Matias en sinds kort ook Bart en Jonas van harte bedanken voor de goede samenwerking, het nalezen van dit proefschrift en de vele interessante vergaderingen, zowel met als zonder pint. Verder bedank ik Ronny om zo goed zorg te dragen voor mijn data. Ook de rest van ons verdiep en de hele PARIS onderzoeksgroep maakten dit onderzoek mee mogelijk.

Wetenschappelijk onderzoek beperkt zich echter niet tot één onderzoeksgroep. Ik kreeg dan ook de kans om mijn werk voor te stellen op internationale conferenties en tegelijkertijd een stuk van de wereld te zien. Bovendien maakte dit het mogelijk om mijn ideeën uit te wisselen met de besten uit het domein. Een aantal van hen werd bereid gevonden in mijn examencommissie te zetelen.

Christian Collberg, Erik D'Hollander, Koen De Bosschere, Daniël De Zutter, Mariusz Jakubowski, Stefan Katzenbeisser, Eric Laermans, Bart Preneel and Jan Van Campenhout, I am very honored to have you all on my PhD committee. Thank you for the time you were willing to spend and the enlightening discussions.

Het professioneel hoogtepunt van de laatste vier jaar was zonder twijfel de stage, inclusief barbecue bij Bill Gates, bij Microsoft Research. Ik wil Bart Preneel in het bijzonder bedanken voor het leggen van de eerste contacten.

Ramarathnam Venkatesan and Mariusz Jakubowski, thanks a lot for having me over at your research group. I had a wonderful time, both professionally and personally. I look forward to our next meeting.

Gent, januari 2008
Bertrand Anckaert

Samenvatting

De snelle verspreiding van de PC is ongetwijfeld in de hand gewerkt door de openheid van het platform. Diezelfde openheid zorgt er ook voor dat software kwetsbaar is.

Dit is een bron van bezorgdheid omdat software steeds vaker waardevolle informatie bevat of beschermt. Medische en financiële informatie wordt bijvoorbeeld meer en meer digitaal opgeslagen. De intellectuele eigendom vervat in film en muziek wordt steeds vaker beschermd door softwaregebaseerde systemen. Bovendien willen sommige gebruikers het gedrag van hun software aanpassen. Zo willen sommigen vals spelen in computerspelletjes. Anderen willen dan weer dat de software functioneert zonder geldige licentie.

Een gebruiker die de volledige controle heeft over het systeem waarop de software draait, kan het programma naar willekeur observeren en wijzigen. In een dergelijke omgeving is het voor een kwaadwillende gebruiker slechts een kwestie van tijd vooraleer de software gekraakt wordt.

Bestaande beschermingsmechanismen verhogen de inspanning vereist voor de eerste succesvolle aanval. Van zodra de eerste succesvolle aanval beschikbaar is, neemt het aantal gekraakte kopieën zeer snel toe. In veel gevallen kan deze aanval immers eenvoudig overgezet worden op andere kopieën.

Dit werk legt de nadruk op het minimaliseren van de gevolgen van zo'n succesvolle aanval. Met behulp van artificiële diversiteit kunnen we de verspreiding van een aanval tegengaan. Dit zorgt ervoor dat informatie verworven uit het aanvallen van één kopie niet zonder meer toepasbaar is op een andere kopie. Bovendien laat het toe om het onderscheid te maken tussen verschillende kopieën. Zo kunnen we de bruikbaarheid van een succesvolle aanval in de tijd reduceren door de toegang tot updates, bijkomende functionaliteit en online diensten te ontzeggen aan gekraakte kopieën.

Een andere toepassing van diversiteit is het coderen van berichten. Door verschillende kopieën te koppelen aan verschillende gebruikers, kan bij het verspreiden van een illegale kopie of een andere inbreuk de dader makkelijker geïdentificeerd worden. Men kan dit het best vergelijken met een digitale vingerafdruk. Een verwante toepassing is steganografie, waarbij een geheim communicatiekanaal wordt opgezet.

Bovendien kan diversiteit gebruikt worden om niet-artificiële verschillen te verbergen. Dit is bijvoorbeeld nuttig wanneer het verschil tussen twee versies van een DRM-applicatie de gebruikersspecifieke sleutel is. Deze sleutel dient geheim te blijven, zelfs voor de rechtmatige eigenaar. Kennis van de sleutel stelt hem immers in staat om de media te ontdoen van de beschermingslaag en die daarna te distribueren. Zonder bijkomende diversiteit kan de locatie van de sleutel gevonden worden door het verschil te nemen tussen twee versies.

Er zijn nog een aantal situaties waarbij het verschil tussen versies gevoelige informatie kan bevatten. Bij digitale vingerafdrukken kan het verschil de plaats van de vingerafdruk verraden, waardoor die makkelijker verwijderd kan worden. In het geval van een beveiligingsupdate kan het verschil tussen de versie voor en na de update een kwetsbaar punt in de software identificeren. Deze informatie kan dan gebruikt worden om een aanval op te zetten tegen verouderde systemen. Door het aanbrengen van bijkomende artificiële diversiteit kunnen de originele verschillen verborgen worden tussen een groter aantal verschillen.

Al deze toepassingen vereisen een manier om semantisch equivalente, maar syntactisch verschillende versies te genereren. Daartoe stellen we een geautomatiseerd diversiteitssysteem voor, gebaseerd op een aantal geparameteriseerde transformaties uit verschillende domeinen: codegeneratie, optimalisatie, obfuscatie, virtualisatie en zelfwijzigende code.

Het aantal versies dat we op die manier kunnen genereren is erg groot. Zo kunnen we tussen $2^{134.150}$ en $2^{1.339.124}$ versies genereren voor de C-programma's uit SPEC CPU2006.

Het aantal versies geeft echter geen volledig beeld. De versies moeten niet enkel verschillend zijn, ze moeten voldoende verschillend zijn opdat informatie uit één kopie geen informatie bevat over een andere kopie (op de functionaliteit na). In de praktijk is deze voorwaarde moeilijk te vervullen. Bovendien hebben we een manier nodig om diversiteit te evalueren zelfs als aan deze voorwaarde niet voldaan is. Hiertoe introduceren we een metriek die aangeeft hoe goed de diversiteit een koppelsysteem om de tuin kan leiden.

Een koppelsysteem schat welke paren codefragmenten uit twee versies verwant zijn. Deze schatting wordt dan vergeleken met de referentierelatie, die definieert welke codefragmenten echt verwant zijn. De schatting kan paren codefragmenten bevatten die niet verwant zijn. In dit geval spreken we van valse positieven. De schatting kan ook verwante paren niet bevatten. Dit noemen we valse negatieven. Hoe meer valse positieven en negatieven, hoe slechter de schatting en dus hoe beter de diversiteit.

Dit werk bespreekt een koppelsysteem dat bestaat uit zeven vage rangschikkers. Elke rangschikker gebruikt een verschillende soort informatie: in-

structiesyntaxis, data, controleverloop, dataverloop, het aantal uitvoeringen, systeemoproepen en het tijdstip van uitvoering. De idee is dat, hoewel transformaties bepaalde types informatie kunnen beïnvloeden, de kans klein is dat alle types informatie tezelfdertijd onbruikbaar worden.

De informatie waarop de rangschikkers gebaseerd zijn, wordt verzameld door een dynamisch instrumentatieraamwerk. Het voordeel van een dergelijk raamwerk is dat het geen manuele ontrafeling of programmabegrip vereist. Aangezien we enkel rekening houden met effectief uitgevoerde code, is de verzamelde informatie bovendien accuraat en correct.

De invoer van elke rangschikker bestaat uit twee codefragmenten. Door middel van een waarde uit het interval $[0, 1]$ wordt aangegeven hoe zeker de rangschikker is dat de twee codefragmenten verwant zijn. Uit evaluatie blijkt dat de individuele rangschikkers veel niet-verwante codefragmenten toch als verwant beschouwen. Dit leidt op zijn beurt tot vele valse positieven.

De oorzaak van een groot aantal van de valse positieven is te vinden bij een aantal grote equivalentieklassen. Om te vermijden dat deze tot een groot aantal valse positieven leiden, kunnen we het aantal relaties per codefragment beperken. Een tweede bescherming tegen valse positieven bestaat erin om de context van de codefragmenten te verruimen. De kans dat twee instructies schijnbaar identiek, maar niet-verwant zijn is een stuk groter dan de kans dat twee basisblokken dat zijn.

Tot slot kunnen we de schatting verbeteren door het combineren en itereren van rangschikkers. Door combinatie kunnen we verschillende types informatie tegelijkertijd beschouwen. De kans dat twee niet-verwante codefragmenten schijnbaar identiek zijn met betrekking tot één type informatie is significant groter dan de kans dat ze identiek zijn met betrekking tot verschillende types informatie tegelijkertijd. Door iteratie kunnen we dan weer eerder verzamelde informatie gebruiken om de schatting uit te breiden of te reduceren. Zo kunnen we bijvoorbeeld paren van codefragmenten identificeren die eerder ten onrechte als verwant werden beschouwd indien die in de controleverloopgraaf niet in de nabijheid van andere verwante codefragmenten zitten.

Het doel van een diversiteitssysteem is dan weer om het koppelsysteem om de tuin te leiden. Er zijn twee mogelijkheden om dit te doen. Ten eerste kunnen we de uitvoer van het koppelsysteem minder bruikbaar maken door het aantal valse positieven en negatieven op te drijven. Daarnaast kunnen we er voor zorgen dat de uitvoer niet binnen redelijke tijd gegenereerd kan worden.

Een eerste reeks van diversifiërende transformaties (gebaseerd op codegeneratie, optimalisatie en obfuscatie) heeft als hoofddoel om het aantal valse negatieven te verhogen. De invloed van de individuele transformaties op het koppelsysteem is vrij beperkt. Zo wordt slechts 24% van de verwante co-

defragmenten niet geïdentificeerd en is hoogstens 37% van de geschatte verwantschappen verkeerd. Wanneer we echter de verschillende transformaties combineren, kunnen we ervoor zorgen dat iets meer dan 3 van de 4 verwante codefragmenten niet langer gedetecteerd worden, terwijl bijna 6 van de 10 geschatte verwantschappen fout zijn.

Met behulp van zelfwijzigende code proberen we het koppelsysteem te vertragen. Meer bepaald zorgen we ervoor dat het verzamelen van informatie, die door de rangschikkers wordt gebruikt, bemoeilijkt wordt. Door het gebruik van zelfwijzigende code kunnen we ervoor zorgen dat de vaak gemaakte veronderstelling dat code constant is niet langer geldt. We introduceren een voorstelling voor zelfwijzigende code die het mogelijk maakt om fijnkorrelige zelfwijzigende code te introduceren, te analyseren en te lineariseren. Op basis hiervan bespreken we twee transformaties die gebruik maken van zelfwijzigende code.

Door de aanwezigheid van zelfwijzigende code dient het instrumentatieraamwerk elke schrijfoperatie te controleren om na te gaan of ze niet naar eerder geïnstrumenteerde code schrijft. Dit alleen vertraagt de uitvoering met een factor twee. Bovendien moet desgevallend code geïnvalideerd en geherinstrumenteerd worden. Dit leidt tot een bijkomende vertraging van meer dan een factor 200 door de toegepaste transformaties.

De laatste transformatie die besproken wordt, is virtualisatie. De idee is om het programma te herschrijven in een unieke instructieset en het te verdelen met een bijhorende virtuele machine. Hierdoor wordt de originele code niet langer uitgevoerd. De originele code is nu data geworden die geïnterpreteerd wordt door de virtuele machine. Hierdoor ontsnapt de code aan het koppelsysteem dat enkel verwante code (en niet verwante data) probeert te identificeren.

Bovendien resulteert de vrijheid van het zelf definiëren van de instructieset in een grote ontwerpruimte en dus veel potentieel voor diversiteit. Zo kunnen we zelf de semantiek van de instructies vastleggen en de manier waarop die gecodeerd worden. Bovendien dienen we ons niet te houden aan het traditionele uitvoeringsmodel.

De voornaamste bijdragen van dit werk zijn het motiveren van de toepasbaarheid van diversiteit binnen de context van een kwaadwillende omgeving, het voorstellen van een praktisch bruikbare metriek voor de evaluatie van diversifiërende transformaties en de introductie van een diversiteitssysteem. Voor dit diversiteitssysteem bestuderen we als eerste de bruikbaarheid van bestaande transformaties uit verschillende domeinen binnen de context van diversiteit. Tot slot hebben we een model geïntroduceerd voor zelfwijzigende code.

Abstract

Openness has arguably been an enabler for the widespread proliferation of the PC (Personal Computer) platform. At the same time, it has left software vulnerable.

This is a growing concern as software more and more contains or controls access to valuable information. For example, medical and financial information may be stored digitally. The intellectual property of movies, music and pictures may be protected by software-based DRM (Digital Rights Management) systems. Furthermore, some users modify software to alter the behavior intended by the provider. For example, they may circumvent copy restriction mechanisms or they may modify a game to become invulnerable.

When the adversary has full control over the environment running the software, he can inspect and modify the software at will. In the presence of such a malicious host, it is only a matter of time before the software is broken.

While most defenses are about delaying the first attack, we focus on minimizing the impact of a successful attack through diversity. As such, this work is an acknowledgment that attacks will continue to emerge. It is useful in business models that can tolerate a limited number of (temporarily) cracked copies.

Without diversity, the number of attacked copies increases very fast as soon as one attack is crafted. This attack can be reused quickly and easily because it does not have to guess much about other targets; they are essentially the same. By introducing artificial diversity, we can delay the spread of an attack. As a result, information learned from one copy may not be applicable to other copies and automated attacks may only be successful on one or a limited number of copies. Furthermore, diversity allows us to discriminate between different versions. Therefore, we can limit the usability of an attack over time by no longer providing access to updates, extra functionality, online services and content to copies that are known to be compromised.

Another application of diversity is message encoding. By linking different versions to different users, traitor tracing can be facilitated. This technique is referred to as fingerprinting. Alternatively, diversity can be used for steganography, i.e., as a covert communication channel.

Diversity can be used to hide the difference between versions as well. In a number of scenarios, different versions are distributed to the public. The difference between those versions may contain personal or security-sensitive information. In the case of fingerprinting, it may reveal the location of the identification, which in turn facilitates removing the fingerprints. In the case of security patches, it may reveal the location of a vulnerability. This information can subsequently be used to exploit unpatched systems.

Each of these applications requires a way to generate syntactically different, but semantically equivalent versions. We therefore present an automated diversity system, composed of parameterized transformations from different domains: code generation, code optimization (factorization and inlining), obfuscation, self-modifying code and virtualization.

The number of versions that we can generate by combining the different transformations is huge for real-life programs. Experimental evaluation shows that the range is between $2^{134,150}$ and $2^{1,339,124}$ for the C benchmarks of the SPEC CPU2006 benchmark suite.

However, range does not tell the whole story. The copies should be more than just different. Ideally, they are sufficiently different so that information acquired from one copy does not reveal any information about other copies other than what could be learned from observing only the I/O (Input/Output) behavior.

In practice, this requirement is hard to fulfill. Furthermore, we need a way to evaluate systems that, even though not theoretically secure, do increase the diversity between versions. The metric we introduce measures how successful the diversifying transformations are in thwarting a matching system.

A matching system estimates which pairs of code fragments from two versions are related. This estimate is then compared to the reference mapping, which defines pairs of related code fragments. The estimated mapping may contain pairs of code fragments which are not related, resulting in false positives. Conversely, the estimate may fail to identify pairs from the reference mapping, referred to as false negatives. A higher false positive and false negative rate indicate a less accurate mapping, and hence a higher level of diversity.

The matching system should be seen as the first automated step in a practical collusion attack against diversity systems. Firstly, it can be used to find correspondences between versions despite the introduced artificial diversity. This can then be used to generalize an attack against one version to other versions. Secondly, it can be used to separate artificial differences from inherent differences. This is useful for an attacker who wants to learn personal or security-sensitive information from the original difference.

We present a matching system that is composed of seven fuzzy classifiers. Every classifier operates on a different type of information: instruction syntax, data, control flow, data flow, execution count, system calls and first time executed. The idea is that, even though transformations may have affected one or more types of information, they are unlikely to have affected all of the types of information at the same time.

The information is collected using a dynamic instrumentation framework. The advantage of such a framework is that it does not require any reverse engineering or program understanding. The instrumented code is generated on the fly, while the original program is used for data accesses. The thus collected information is guaranteed to be accurate and correct, as there is no uncertainty about the code that is actually executed.

Each of the classifiers takes as input two code fragments and indicates its confidence that the two code fragments are related by returning a value in the interval $[0, 1]$. The evaluation of each of these classifiers in isolation reveals that many pairs of unrelated code fragments are considered to be related, which leads to high false positive rates.

A number of large equivalence classes are the root cause of many false positives. One possible countermeasure is to limit the number of matches per code fragment, thereby reducing the impact of large equivalence classes on the false positive rate. A second countermeasure is to widen the scope of the code fragments under consideration. The probability of observing two seemingly identical, but unrelated instructions is significantly higher than the probability of observing two seemingly identical, but unrelated basic blocks.

Finally, we can improve the estimated mapping by combining and iterating classifiers. Through combination, we will take different types of information into account at the same time. The probability that two unrelated code fragments appear identical with respect to one type of information is significantly smaller than the probability that they appear similar with respect to multiple types of information at the same time.

Through iteration, we can build upon already obtained information to extend or filter the existing mapping. For example, code fragments that have been falsely assumed related in an earlier iteration may be detected in subsequent iterations when it becomes apparent that no related code fragments are within their proximity in the control flow graph.

The goal of diversity, however, is to thwart a matching system. This can be done in two ways. Firstly, we can render the output of the matching system less usable by increasing the false negative or false positive rate. Secondly, we can prevent the matching system from producing the output within reasonable time.

A first series of diversifying transformations (based on code generation, optimization and obfuscation) is targeted primarily at increasing the false negative rate. Through these diversifying transformations, we try to make related code fragments seemingly different.

The impact of the individual transformations on a particular matching system is relatively modest: the highest observed false negative rate is 0.24 and the highest false positive rate 0.37. However, when we combine the different transformations, we can inflict a false negative rate of 0.76. This means that over 3 out of 4 related code fragments are not detected. The false positive rate is at 0.58, meaning that almost 6 out of 10 reported matches are incorrect.

In a second approach we try to delay the matching system, in particular the collection of the information used by the classifiers. By introducing self-modifying code, we can undermine the commonly made assumption that code is constant. We introduce a representation for self-modifying code which allows for the generation and analysis of fine-grained self-modifying code. Two diversifying transformations built on top of this model are discussed.

Due to the presence of self-modifying code, the dynamic instrumentation framework needs to monitor every write instruction to see if it affects any previously instrumented code. This alone results in a slowdown of a factor 2. If the write operation affects already instrumented code, the code needs to be invalidated and re-instrumented. This leads to an additional slowdown of a factor of over 200 because of the introduced fine-grained self-modifying code.

The last transformation we discuss is virtualization. The idea is to rewrite the entire program in a custom instruction set and to ship it with an implementation of a virtual machine that interprets the instruction set. As a result, the original code of the program is no longer directly executed. It is now data that is interpreted by the virtual machine. Therefore, it will no longer be considered by the matching system which only tries to match code, and not data.

Furthermore, the freedom to design our own instruction set architecture results in a large design space, and hence a lot of potential for diversity. We get to choose the semantics of the instructions and their encoding, and we can abandon traditional execution models.

The main contributions of this thesis are the motivation of the applicability of diversity in the malicious host model, the introduction of a practical metric to evaluate the success of diversifying transformations, and the presentation of a practical diversity system. We are the first to extensively study the applicability of existing transformations from different domains within the context of diversity. Finally, we have introduced a model which facilitates manipulating self-modifying code.

Contents

List of Tables

List of Figures

List of Acronyms

API Application Programming Interface

ASLR Address Space Layout Randomization

CFG Control Flow Graph

CISC Complex Instruction Set Computer

CPU Central Processing Unit

CTS Code Transformation Signature

DRM Digital Rights Management

IA-32 Intel Architecture – 32 bit

ISA Instruction Set Architecture

MSIL Microsoft Intermediate Language

PC Program Counter

RISC Reduced Instruction Set Computer

ROC Receiver Operating Characteristic

SE-CFG State Enhanced Control Flow Graph

TPM Trusted Platform Module

VM Virtual Machine

Prologue

'Menelaus,' replied she, 'I will make it all quite clear to you. There is an old immortal who lives under the sea hereabouts and whose name is Proteus. He is an Egyptian, and people say he is my father; he is Neptune's head man and knows every inch of ground all over the bottom of the sea. If you can snare him and hold him tight, he will tell you about your voyage, what courses you are to take, and how you are to sail the sea so as to reach your home. He will also tell you, if you so will, all that has been going on at your house both good and bad, while you have been away on your long and dangerous journey.'

'The moment you see that he is asleep seize him; put forth all your strength and hold him fast, for he will do his very utmost to get away from you. He will turn himself into every kind of creature that goes upon the earth, and will become also both fire and water; but you must hold him fast and grip him tighter and tighter, till he begins to talk to you and comes back to what he was when you saw him go to sleep; then you may slacken your hold and let him go; and you can ask him which of the gods it is that is angry with you and what you must do to reach your home over the seas.'

Then we rushed upon him with a shout and seized him; on which he began at once with his old tricks, and changed himself first into a lion with a great mane; then all of a sudden he became a dragon, a leopard, a wild boar; the next moment he was running water, and then again directly he was a tree, but we stuck to him and never lost hold, till at last the cunning old creature became distressed, and said, 'Which of the gods was it, Son of Atreus, that hatched this plot with you for snaring me and seizing me against my will?
What do you want?'

After "The Odyssey" by Homeros, translated by Samuel Butler

1
Introduction

In nature, genetic diversity provides protection against an entire species being wiped out by a single virus or disease. The same idea applies to software, with respect to resistance to the exploitation of software vulnerabilities and program-based attacks [van Oorschot 03]. However, the concept of diversity has not yet been fully embraced by the software community.

In many other domains, however, risk diversification is widely accepted as good practice. In agriculture, for example, farmers tend to grow more than one crop, a lesson learned hard over time. In Ireland, e.g., one particular variety of potato called "the lumper" became the dominant harvest for the country by 1840, making up the only significant source of food for about 3 million people. In 1845, a fungus blighted the crops, and more than 1 million people died of malnutrition or starvation within two years. According to historians, the catastrophe led to the diversification of Irish crops and the number of acres devoted to lumper potatoes dropped from 2 million to 300,000 in the two years following the famine.

In the financial world, most investment professionals agree that diversifying the portfolio is the most important component to reach long-range financial goals while minimizing risk. Not putting all your eggs in one basket is even conventional wisdom.

Software, on the other hand, is surprisingly homogeneous: consider the very small number of different Internet browsers currently in use; and the number of major operating systems in use, which is considerably smaller than 15 years ago.

While this clearly facilitates the development and maintenance of compatible software and content, several prominent security engineers have warned about the dangers of the lack of diversity in software. In a controversial report *Cyber Insecurity: The Cost of Monopoly* [Geer 03], the authors claim that the absence of diversity greatly increases the risk of a single attack compromising the entire installed base. They argue that risk diversification is a primary defense against aggregated risk when that risk cannot otherwise be addressed.

Real-life illustrations of this risk are numerous, the NIMDA and Slammer worms have attacked millions of Windows-based computers. The main reason is that these worms do not have to guess much about the target computers because nearly all computers have the same vulnerabilities.

1.1 Diversity to Minimize the Impact of an Attack

Securing software against malicious code is, for the most part, a problem that has been solved in theory. Yet, in practice, vulnerabilities continue to be discovered at an astonishing rate. Buffer overflows, for example, were a solved problem as early as the 1960s and continue to be the most common type of security problem [Park 04].

Due to the complexity of modern software and the increasing body of legacy code, this and other types of vulnerability continue to exist. Software diversity as a security mechanism against malicious code attacks was proposed by Cohen [Cohen 93] under the term *program evolution*. A number of techniques have been presented since, of which address space layout randomization is probably most widely applied. Address space layout randomization randomizes the location of the stack, heap and shared libraries to fortify systems against the exploitation of memory-related errors. In a way, it is an acknowledgment that buffer overflow and related types of attack will continue to emerge. The approach tries to mitigate the problem by removing predictability and consistency between different executions. The technique has made its way into widely distributed operating systems, e.g., Mac OS X 10.5, Linux via PaX and Windows Vista.

Protecting software against a malicious host is, in some cases, a theoretically unsolvable problem [Barak 01]. Intuitively, any protection scheme other than a physical one depends on the operation of a finite state machine, and ultimately, given physical access, any finite state machine can be examined and modified at will, given enough time and effort [Cohen 93].

1.1.1 Incentives for Tampering

The incentive to tamper with software originates from the difference between the behavior intended by the software or content provider and the behavior

desired by the user. There are many situations where these two differ. Tampering is thus about transforming the semantics intended by the provider to the semantics desired by the user.

Software Piracy

Some software will not install without a valid license key. Other software stops working after an evaluation period. For some users, this is undesired behavior.

The behavior intended by the software provider is often enforced through defense mechanisms such as license keys or activation schemes. Tampering with these defense mechanisms can turn the behavior intended by the software provider into the behavior desired by the end user.

In practice, these mechanisms continue to be broken on a regular basis. If users are able to use the software without compensation, this can impact the revenues of the software provider. One study, by the Business Software Alliance, estimates the value of illegally installed software at US$40 billion for 2006 [BSA 05]. Note that the accuracy of this number is doubtful, as the Business Software Alliance represents a number of the world's largest software providers and may have an incentive to overestimate.

Cheating in Games

Games are a type of software that is susceptible to piracy as well. Additionally, tampering with games can be used to cheat. This is an increasing concern for game developers as more and more games are multiplayer games. If one or more players cheat, this ruins the experience of the honest players. Ultimately, this could lead to reduced interest in the game and thus impact the revenue of the gaming industry.

Furthermore, the value of virtual characters and assets in massively multiplayer online games is growing. For example, a virtual space resort in the game Entropia Universe sold for the equivalent of US$100,000[1]. It should be clear that this creates a big incentive to cheat.

Circumventing Content Restrictions

Digital containers are used more and more to provide controlled access to copyrighted material. Currently, music distribution is a high-profile application, with Apple's FairPlay, Microsoft's Windows Media and Real's Helix Digital Rights Management (DRM) systems. These DRM systems impose restrictions on the usage of media. Many users do not like these restrictions. In this type of system, software is often the weakest link. If that link is broken,

[1] http://news.bbc.co.uk/1/hi/technology/4953620.stm

copyrighted material may be distributed illegally, again eliminating potential revenues. The value of illegally obtained music is estimated at US$4.2 billion by the RIAA and the value of illegally obtained video is estimated at US$6.1 billion according to the MPAA. As the RIAA represents the recording industry and the MPAA the movie industry, these numbers should be treated with care.

Another example can be found in Adobe Acrobat where, e.g., the printing or the copying of content can be prohibited. As the user can still view the content, these are clearly artificially introduced restrictions which some users find annoying. If the restrictions can be circumvented too easily, the sales of Adobe's software may decrease as this particular feature becomes useless.

When the incentive to tamper is as large as illustrated in the previous examples, we should no longer expect to build one super-strong defense that will withstand attack for an extended period of time. Even hardware solutions are not safe [Anderson 96]. In the arms race between software protectors and attackers, the one that makes the last move often has the winning hand. Acknowledging this might be a first important step for software providers. If we accept that a system will be broken, the remaining defense is to minimize the impact of a successful attack. We need to make sure that an attack has only local impact, both spatially and temporally. Reducing the impact of an attack might furthermore reduce the very incentive that leads to attacks.

1.1.2 Related Software Protection Techniques

The financial losses mentioned above serve as incentives to make software more tamper resistant. Consequently, both industry and academia have conducted a lot of research in this area. Most defenses against malicious hosts are about delaying the first attack. The success of these techniques varies in terms of the additional time and effort required by a tamperer.

Most studies have looked at the issue of software piracy, possibly because it affects the revenues of software providers most directly. Among the approaches that have been explored in recent history to address the problem of tampering with software are legal, ethical and technical means.

Legal means are based on the fear of consequences of violating content protection laws. But while in most cases the legal means are available, prosecution on a case by case basis is economically infeasible. Furthermore, it is conceived as bad publicity and can take a long time.

Ethical measures relate to making software tampering morally unappealing. While the intentions are laudable, it takes even more effort and time to change the moral standards of a large group of people.

The existing technical means almost all have a static nature of defense, in which a protection mechanism is built into the distributed software. Once this

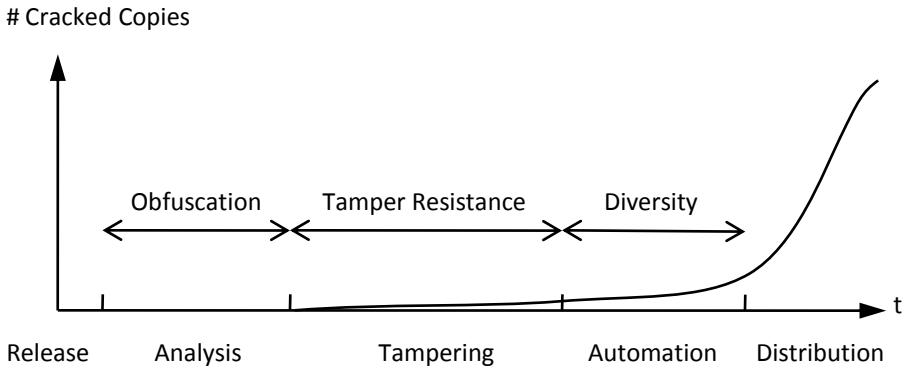


Figure 1.1: Techniques to delay tampering: obfuscation, tamper resistance and diversity

protection is broken, no further steps can be taken to protect the software or content.

Software-only Technical Defenses

Release Figure 1.1 shows the relation between diversity and the two other main technical software protection techniques: obfuscation and tamper resistance. This simplified diagram shows the typical evolution of the number of tampered copies of a program. First, the attacker needs to comprehend at least some portion of the program so that he can proceed to the phase of intelligent tampering. The tampering is initially done manually through trial and error. Once the attacker has successfully changed the behavior of the software, he will try to automate the attack and share it with others. Once the attack is automated and publicly available, the number of cracked copies increases significantly, as it is now accessible to people without a thorough technical background. Each of the software protection techniques operates on a different aspect of the crack cycle: obfuscation complicates analysis, tamper resistance complicates modification and diversity complicates automation.

Software fingerprinting [Collberg 99, Venkatesan 01] does not prevent the copying of software as such, but dissuades the pirate by increasing the likelihood of being caught. It can be seen as a technical means to facilitate legal protection.

One advantage of fingerprinting over most other copy protection techniques is that it is more difficult for an attacker to be sure that he has removed a fingerprint, than it is to be sure that a copy protection mechanism has been cracked. Whereas the latter can often easily be tested, i.e., a copy works or does

not work, determining whether or not the fingerprint is still present is harder, and hence a pirate can modify a fingerprinted program at will, but when he wants to use or redistribute it illegally, the fear that he could be identified remains. On the other hand, fingerprinting comes with the obvious disadvantage of its reliance on cumbersome legal measures.

Like fingerprinting, software aging [Jakobsson 02], increases the likelihood of the pirate being caught. This technique relies heavily on program updates, which are crafted to ensure that more recent versions can read the output of older versions, but not vice versa. For this technique to work, it is assumed that illegitimate users interact only with the original pirate to obtain these updates. As a result, either pirated software becomes increasingly unusable because it is not kept up to date, or the interaction with the original pirate increases and as a result the likelihood of him being caught increases as well. With software aging, what a legitimate user pays for is the guarantee of continuing access to updates. This protection mechanism can therefore be seen as a dynamic form of protection.

Software as a Service

The vulnerability of software originates from the openness of the PC architecture. Therefore, the most straightforward way of protecting software is not to run it on a PC accessible to the attacker, but on a server. As an example, consider the closely guarded search algorithm of Google.

Software as a service is a strong protection against analysis and tampering. There is little more an attacker can learn about the software than its I/O behavior and timing information. Furthermore, he cannot modify the software as it resides on an external server.

This approach also has some drawbacks: it requires a connection to be able to use the service and may lead to privacy concerns. Furthermore, it is not useful for all forms of software and content protection. For example, the restriction that one can view the document but not print it will still need to be enforced on the local PC.

Hardware-assisted Technical Defenses

All hardware-based approaches we are aware of use tokens. In these schemes, it is impossible to execute the program without the presence of a trusted hardware component, such as a specific CD, dongle, smart card, etc.

One approach [Kent 81] involves using encrypted programs, with instructions decrypted immediately prior to execution. As the decrypted instructions never leave the trusted hardware, inspection and analysis of the code itself is very hard.

One issue that is not addressed by this approach is the extraction of useful information gained by an adversary examining the memory access patterns of executing programs. Therefore, Goldreich and Ostrovsky [Goldreich 96] have reduced the problem of software protection to that of efficient (in the theoretical sense) simulation on oblivious RAMs. Oblivious RAMs replace instruction fetches in the original program by sequences of fetches, effectively randomizing memory access patterns to eliminate information leakage.

The Trusted Computing Group [Felten 03] was started by major players in the hardware and software market such as AMD, Intel, IBM and Microsoft. It advocates the adoption of the Trusted Platform Module (TPM), a low-cost hardware module that provides cryptographic services and a hardware random number generator. It furthermore includes support for capabilities such as remote attestation and binding. Remote attestation creates an unforgeable summary of the hardware, boot, and host operating system configuration of a computer, allowing a third party (such as a digital music store) to verify that the software has not been changed. Binding can be used to make sure that critical data is only accessible by a single platform when the conditions (e.g., software configuration) specified in the binding are met. Currently, an estimated 70-80 million PCs have a TPM. However, it is not clear how many TMPs have been activated.

Being able to rely on a hardware-protected foundation of trust will certainly facilitate the protection of software and content, especially when combined with secure I/O. Unfortunately, there are some causes for concern. To name a few: the chain of trust may be too long to verify, the number of valid configurations may be impractically large, it can facilitate user lock-in and it raises privacy issues. We will not elaborate on these issues as they are beyond the scope of this dissertation, but we expect to see many interesting developments in this domain in the future.

1.1.3 Break Once Break Every Time Resistance

Diversity is a promising additional layer of defense, as, rather than trying to postpone the first attack, it is about limiting the impact of a successful attack in space and time. If we can sufficiently diversify the installed base, an attack against one instance will not compromise other instances (spatially renewable defense). If we can furthermore discriminate between genuine and tampered-with copies when updating software (temporally renewable defense), an attacker will ultimately need to revert to time-consuming manual reverse engineering and tampering to craft a specific attack for every instance and every update. This is discussed in more detail in Section 2.1

1.1.4 Break Once Run Every Time Resistance

Unfortunately, diverse copies are in conflict with the current software distribution model, which requires identical copies to leverage the near zero marginal cost of duplication. Not surprisingly, commercial implementations of this technique can be found in situations where a network connection can be assumed to distribute the copies digitally: DRM implementations for on-line stores and digital broadcasters [Zhou 06].

In an alternative, orthogonal approach, we try to combine the best of both worlds by introducing diversity after the distribution. The run-time execution of the code is diversified based upon additional chaff inputs (such as time and hardware identifiers) and variable program state, including additional fake input dependencies. The goal is twofold: (i) to make it harder for an attacker to zoom in on a point of failure and (ii) to limit the impact of a successful attack to a short period of time, a particular computer, a subset of the input space, etc. We elaborate on this approach in Section 2.2.

1.2 Diversity to Encode Messages

The ability to generate different versions of a program has other applications as well. We can, for example, associate a different message to each version. Embedding messages within a program has a number of applications. Firstly, it can be used to embed additional information about the program itself, such as debug information or branch hints for the processor. Secondly, it can be used to exchange secret messages. Thirdly, it can be used to embed information about the buyer or version of the software, e.g., to facilitate the tracing of illegitimate sharing. We will discuss the latter two applications in more detail.

1.2.1 Steganography

Steganography embeds a secret message in a seemingly innocuous cover object. Certain governments and police agencies claim that widely available encryption software could make wiretapping more difficult and therefore try to restrict the strength of encryption algorithms. Citizens seeking privacy could use steganography as an alternative to conceal their communication. While cryptography is about protecting the content of messages, steganography is about concealing their very existence [Katzenbeisser 00].

Digital cover objects most often are media, such as image and music files, that contain noise and are perceived by imperfect human senses. As a result, they contain many redundant bits, which can be modified to embed secret messages.

We have looked at the largely unexplored field of steganography for programs. This differs significantly from steganography for media because changing as little as a single bit of a program can cause it to fail entirely. Hence different techniques are required for embedding messages in programs.

Our prototype implementation can achieve an encoding rate of $1/26.96$, four times the encoding rate of the previous state-of-the-art prototype tool Hydan $(1/110)$ [El-Khalil 04]. Without countermeasures, neither the messages embedded by our tool nor by Hydan are stealthy. When taking countermeasures to embed the messages more stealthily, our encoding rate drops to $1/88.76$. Section 4.7.5 contains a more extensive discussion.

1.2.2 Fingerprinting

Like steganography, fingerprinting embeds a message into a cover object. However, there are a number of important differences:

- An active warden is assumed: the attack model assumes that attempts will be made to remove the fingerprint. Therefore, the fingerprint needs to be resilient against attacks.
- The existence of the message does not need to be a secret, as long as this does not compromise the resilience.
- The message is typically more strongly related to the cover object: it typically includes vendor, product, and customer identification numbers, whereas, in the case of steganography, the message can be completely unrelated to the cover object.
- It is valid to assume that the original object is available during the extraction.

If the message can be reliably retrieved, and if it has a mathematical property that allows us to argue that its presence is the result of deliberate actions, it may enable theft detection, traitor tracing and legal actions against copyright violators.

Fingerprinting objects makes them vulnerable to collusion attacks. An adversary might attempt to gain access to several fingerprinted copies of an object, compare them to determine the location of the fingerprints, and, as a result, be able to make them unrecognizable. As we will see in the next section, artificial diversity can be used to mitigate this type of attack.

1.3 Diversity to Hide the Difference between Versions

In a number of situations software providers release different versions of software to the public. In some cases, the difference between versions reveals security-critical information. Diversity can be used to hide the difference between those versions within a large number of artificial differences.

1.3.1 Hiding Version-specific Information

A first example has been discussed in the previous section: fingerprinting. In this case, the different versions are released more or less simultaneously. The difference between two versions may reveal the location of the fingerprint. Finding the exact location of the fingerprint by comparing two or more versions greatly facilitates removing or distorting it.

As a second example, consider a software implementation of a DRM system, e.g., iTunes. This type of software is responsible for enforcing certain restrictions, e.g., content can only be accessed by one user. As a result, the software needs to behave differently for different users. This is often enabled through a user-specific cryptographic key.

If that key is the only difference between two versions, a simple comparison can be used to locate the information within the program. As a result, it is vital to diffuse the keys over the entire code and to hide the relevant code snippets.

1.3.2 Patches

A second example is that of software patches. In this case, different versions are released over time. Software providers rarely succeed in creating flawless software the first time around. The advent of the Internet has facilitated fixing some of these flaws after the initial release. As a result, security holes and other bugs are more and more fixed through patches.

In the case of security patches, the difference between a patched and an unpatched version allows an attacker to pinpoint the vulnerable part of the software. The difference is often small, as security updates often patch a security leak through local changes (e.g., in a single module or function). This greatly reduces the time to find a vulnerable part of the software and thus facilitates the creation of an attack against unpatched versions. In practice, there is a relatively large time frame between the release of a patch and the time that most systems are actually patched. Some systems never get patched. The term Exploit Wednesday has been coined in the context of exploiting vulnerabilities fixed by released patches, referring to Patch Tuesday, the second Tuesday of the month, when patches are typically released by Microsoft.

1.4 A Metric for Diversity

One of the hard problems in software protection is obtaining a measure for the strength of techniques. The holy grail of determining a provable and useful lower bound on the workload required to attack software is in many practical settings still a distant dream. Nevertheless, as we highly value quantitative evaluation, we set out to define a metric for diversity. This metric, albeit imperfect, does provide an indication of the diversity introduced by transformations. We consider this metric to be one of the main contributions of this dissertation.

As the saying goes, we need to set a thief to catch a thief. Therefore, we have set up a collusion attack against diversity systems. In a collusion attack, two or more versions are compared against one another in order to break the system. This type of attack is specific to systems that generate more than one version and, as such, has not been dealt with in other approaches to tamper resistance. One notable exception is fingerprinting, where every user gets a unique version with embedded information to enable traitor tracing. However, no publicly available research explicitly deals with this type of attack.

Perfect protection against collusion attacks would mean that the versions are *sufficiently* diverse to make sure that information learned from one version does not facilitate attacking another version. The applicability of information learned from one version to other versions is hard to measure. Therefore, we will use an approximative metric based on the false positive and false negative rate of a matching system. In our definition, the output of a matching system is a set of pairs of code fragments (instructions or basic blocks) from version A and version B which are considered to be related. Such a matching system can be used as a first step in the generalization of attacks and the separation of artificial differences from true differences.

Intuitively, the false negative rate indicates the fraction of related code fragments that was not recognized by the matching system. The false positive rate indicates the fraction of unrelated code fragments in the reported mapping. As such, we believe that the false negative rate is more important: if an attacker is looking for a related code fragment, a false negative rate of 0.5 means that he has a fifty percent chance of not finding it at all in the reported mapping. On the other hand, if it is in the reported mapping, but with a false positive rate of 0.5, the incurred penalty is that he has to manually check or try two candidates.

Our metric assumes that a higher false positive and false negative rate indicate a higher level of diversity, as this indicates more success in thwarting a collusion attack.

1.4.1 Related Work

It has been understood for long that traditional, text-based matching systems are insufficient to match sequences of symbols in the language of computer programs.

Therefore, most related work in this area has focused on graph matching techniques to match related portions of the code. While this has proved to be useful for domains where the versions have not been made different deliberately, obfuscating transformations such as control flow flattening [Wang 01] are capable of foiling these techniques.

This work is closely related to recent work by Zhang et al. [Zhang 05]. However, to the best of our knowledge, this is the first time this type of technique has been applied to deliberately diversified versions of a program.

1.4.2 The Matching System

Our matching system is built on top of a number of fuzzy classifiers. These classifiers indicate their confidence that two code fragments are related based upon different types of information. The advantage is that we do not limit ourselves to one specific type of information (e.g., only textual or control flow information). The framework can be readily extended to include new types of information.

Fuzzy Classifiers

We have chosen to work with information collected during the execution of the programs. As such, this information is guaranteed to be correct and accurate. The supported types of information are:

Instruction syntax The classifier based on instruction syntax is the most related to text-based matching systems. It scores a pair of code fragments based upon the resemblance of the opcode and operands of the instructions.

Data The classifier based on data compares the values read and written by two code fragments.

Control flow The classifier based on control flow indicates the proximity in the control flow graph to other related code fragments. As such, it is closely related to graph-based matching techniques.

Data flow The classifier based on data flow is similar, but operates on the data flow graph.

Execution count The classifier based on execution count compares the execution count of both code fragments.

System calls The classifier based on system call information will compare the values passed to and returned from system calls.

First time executed The classifier based on first execution time will compare the first time a code fragment was encountered in the trace.

To evaluate the strength of these classifiers, we have applied them to two identical versions of a program. This evaluation shows that there are large equivalence classes: sets of code fragments which are all perceived as identical. If we map all the code fragments of version A from an equivalence class to all code fragments of version B from that equivalence class, we have an unacceptably high false positive rate. A number of countermeasures are needed to have a workable matching system.

Building a Matching System from Fuzzy Classifiers

The most obvious countermeasure is to limit the number of matches per code fragment. This technique will avoid that a small number of large equivalence classes is responsible for a huge number of false positives.

The second countermeasure is to look at a broader context. When looking at two unrelated instructions in isolation, there is a fairly high probability that they appear similar. The probability that two unrelated basic blocks appear similar is significantly smaller. The granularity of code fragments can thus be the basic block level and the instruction level.

The third solution is to look at more than one type of information at the same time. By combining different classifiers, we can again decrease the false positive rate. The probability that two unrelated code fragments appear similar with respect to one type of information is significantly higher than the probability that they look similar with respect to multiple types of information at the same time.

The fourth solution is to iteratively apply the matching system. Through iteration, subsequent phases can build upon the mapping generated by previous iterations to extend or filter the mapping.

1.5 Generating Different Tamper-resistant Versions

In most applications of code translation and code transformation, the goal is to create a single version which has some property the original version did not have, or to create a single version which is optimal with respect to some properties. For example:

- The goal of compilation is to create a version targeted at a particular platform.
- The goal of optimization is to create the smallest, fastest or most energy efficient version.

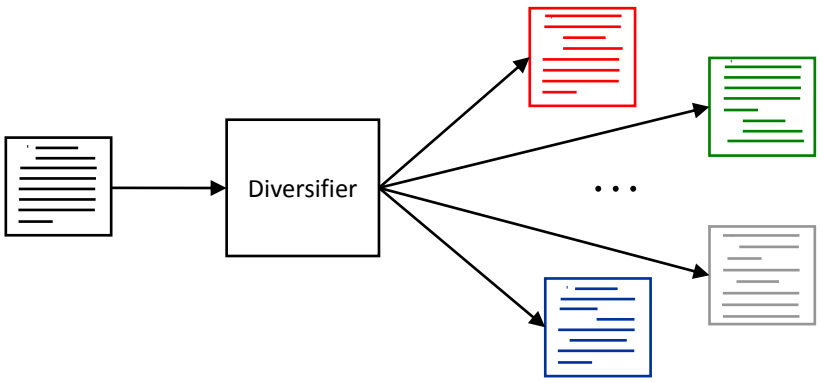


Figure 1.2: Schematic of a diversity system

- The goal of instrumentation is to create a version which records information about the execution.

- The goal of obfuscation is to create the least understandable version.

During these translations and transformations, choices need to be made. The choices are often so abundant that it is infeasible to make the best choice within reasonable time. Instead, heuristics are used to select an alternative which leads to a reasonably good version.

This dissertation explores a number of applications which benefit from not choosing a single alternative each time a choice needs to be made, but from choosing multiple alternatives when choices need to be made to generate different versions. This has been illustrated in Figure 1.2.

The main application is to make sure that information learned from one version is not applicable to other versions. As an approximative metric, we will use the false positive and false negative rate of our matching system. Higher false positive and false negative rates indicate a better diversification.

Experience in matching program versions shows that discriminative invariants between versions serve as signposts. Ideally, every discriminative invariant should be eliminated. In practice, it proves to be difficult to design a single monolithic transformation which removes discriminative invariants for the different types of information at the same time.

However, many transformations each affect one type (or a limited number of types) of information. Other transformations help to undermine assumptions made by the attacker. By combining many of these "smaller" transformations affecting different types of information, we can increase the range and complexity of randomization.

1.5.1 Basic Transformations

We have studied a number of transformations from different domains. A number of code factorization and code inlining transformations have been parameterized to undermine the assumption that there is at most one match per code fragment. We use control flow obfuscating transformations to make the information based on control flow less accurate and slower to evaluate. By randomizing the code generation process, we can partially fool order- and syntax-based matching systems.

Experimental results show that it is easy to generate many syntactically different but semantically equivalent programs. The range of our diversity system is far larger than the estimated number of atoms in the universe[2] (10^{81}) for every non-trivial program. For example, the transformation based on instruction selection alone has a range of $2^{21936} \approx 10^{6606}$ for the smallest C benchmark in the SPEC CPU2006 benchmark suite.

Clearly, the range is not the problem. However, it is not sufficient for the versions to be merely different. We can see that when using the different transformations in isolation, the impact on our metric is relatively small. For an exemplary matching system, the highest observed false negative rate for an individual transformation is around 0.24. When different transformations are combined, with a comparable overall cost, the false negative rate climbs to over 0.76. The highest observed individual false positive rate is 0.37, while the false positive rate for the combined transformations is around 0.58.

1.5.2 Self-modifying Code

The results reported above show that we are heading in the right direction. We can go even further by using self-modifying code. This type of code undermines the commonly made assumption that code is constant. Many tools are not capable of dealing with this type of code, others incur a significant run-time overhead in the presence of self-modifying code.

Self-modifying code has long been applied in an ad hoc manner. We introduce a model which allows for the generation and analysis of self-modifying code, when starting from known code. The main novelty is that all instructions that can ever be executed are represented in a single control flow graph.

Earlier approaches [Dux 05] represent only the code present at a any given point in time, resulting in a plethora of control flow graphs. As such, they do not contain code that (i) was executable in an earlier phase, but is no longer executable, nor (ii) code that may become executable in a later phase. As

[2] Assuming there are a trillion galaxies with sizes comparable to our galaxy, which probably has no more than 10^{69} atoms. This estimate does not count dark matter, brown dwarf stars or dwarf galaxies.

such, none of these graphs is a superset of all possible executions, a property required by many analyses to deal conservatively with all run-time behaviors.

Our model allows us to introduce fine-grained self-modifying code, with the additional advantage that multiple instructions are actually executed at a single address. Most existing applications of self-modifying code are more coarse-grained, e.g., where entire sections are decoded prior to execution, often leaving the entire program in the clear afterwards. When multiple instructions reside at the same address, we are guaranteed that it is never entirely in the clear.

Self-modifying code can be combined with the previously discussed techniques. However, the presence of self-modifying code makes it impractical for our matching system to collect the required information within reasonable time. As such, we were unable to report the observed false positive and false negative rates. We believe an attacker will experience similar setbacks when analyzing self-modifying code, as it is generally assumed to be one of the main problems of reverse engineering [Cifuentes 95].

1.5.3 Virtualization

The last introduced transformation is rewriting the code for a custom virtual machine and shipping it along with an implementation of that virtual machine.

This undermines the basic operation of the matching system, which tries to relate code fragments between different versions. As a result of virtualization, the original code is no longer executed, but becomes data interpreted by the virtual machine.

Furthermore, this approach opens up a whole new range of possibilities for diversity and tamper resistance, as we get to choose our own instruction set architecture and it allows us to abandon traditional execution models.

Experimental evaluation indicates that these techniques, unless optimized significantly, are too expensive to be applied to performance-critical code and should be used sparingly, for example, in DRM and license systems which are less performance-critical.

1.6 Contributions

The contributions presented in this dissertation are:

- We were the first to suggest and study the applicability of diversity in the malicious host model in the academic literature. Past research on diversity has either focused on the malicious code model or is closely guarded as company trade secrets.

- We have designed and implemented a practical and extensible system to match related code fragments between versions. Furthermore, this system provides the first quantitative evaluation of a diversity system.

- We have designed and implemented a practical system to generate a large number of semantically equivalent but syntactically different programs. The resulting versions are able to fool the matching system to a large extent.

- We have defined a model for self-modifying code which allows for the generation and analysis of self-modifying code. Based upon this model, fine-grained self-modifying code can be introduced, which significantly slows down run-time observation.

- We were the first to study the potential of virtualization within the malicious host model.

1.7 Publications

Parts of this dissertation have been published earlier at international conferences.

The idea to use diversity as a defense against piracy was introduced at the Fourth ACM Digital Rights Management Workshop in 2004.

- Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Software piracy prevention through diversity. In *Proceedings of the 4th ACM Workshop on Digital Rights Management*, pages 63–71. ACM Press, 2004. ([Anckaert 04a])

The idea to embed different versions in a single program has been presented at the Second International Workshop on Security in 2007. This paper won the "Best Student Paper Award".

- Bertrand Anckaert, Mariusz Jakubowski, Ramarathnam Venkatesan, and Koen De Bosschere. Run-time randomization to mitigate tampering. In *Proceedings of the 2nd International Workshop on Security*, volume 4752 of *Lecture Notes in Computer Science*, pages 153–168. Springer-Verlag, 2007. ([Anckaert 07a])

The application of diversity for steganography in the context of programs was discussed at the Seventh International Conference on Information Security and Cryptology in 2004 and was published in 2005.

- Bertrand Anckaert, Bjorn De Sutter, Dominique Chanet, and Koen De Bosschere. Steganography for executables and code transformation signatures. In *Proceedings of the 7th International Conference on Information Security and Cryptology*, volume 3506 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, 2005. ([Anckaert 05])

The applicability of virtualization in the context of software protection was published at the Sixth ACM Digital Rights Management Workshop in 2006.

- Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper resistance. In *Proceedings of the 6th ACM workshop on Digital Rights Management*, pages 47–58. ACM Press, 2006. ([Anckaert 06])

The model for self-modifying code has been presented at the Eighth Information Hiding Conference in 2006 and was published in 2007.

- Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In *Proceedings of the 8th Information Hiding Conference*, volume 4437 of *Lecture Notes in Computer Science*, pages 232–248. Springer-Verlag, 2007. ([Anckaert 07b])

The author also contributed to a number of publications not discussed in this dissertation. An alternative application of self-modifying code was published at the Sixth International Workshop on Information Security and Cryptology in 2005.

- Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *Proceedings of the 6th International Workshop on Information Security Applications*, volume 3786 of *Lecture Notes in Computer Science*, pages 194–206. Springer-Verlag, 2005. ([Madou 05a])

A discussion of the interaction of static and dynamic analysis used by an attacker was illustrated in a publication at the Fifth ACM Digital Rights Management Workshop in 2005.

- Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital Rights Management*, pages 75–82. ACM Press, 2005. ([Madou 05b])

A set of metrics from the domain of software complexity have been evaluated in the context of obfuscation to measure the potency of a number of transformations and the effectiveness of deobfuscating transformations. This work has been presented at the Third Workshop on Quality of Protection in 2007.

- Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: A quantitative approach. In *Proceedings of the 3rd Workshop on Quality of Protection*, pages 15–20. ACM Press, 2007. ([Anckaert 07c])

Finally, the result of the author's master thesis was presented at the Tenth Euro-Par Conference in 2004. This thesis was our introduction to link-time binary rewriting. Here, the goal was to optimize IA-64 programs.

- Bertrand Anckaert, Frederik Vandeputte, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Link-time optimization of IA64 binaries. In *Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 284–291. Springer-Verlag, 2004. ([Anckaert 04b])

1.8 Outline

This dissertation is organized as follows. Chapter 2 contains a more detailed discussion of the application of diversity to minimize the impact of an attack. The matching system is introduced in Chapter 3. The diversity system is introduced and evaluated with respect to a matching system in Chapter 4. The more advanced transformations, self-modifying code and virtualization, are the topic of Chapter 5. In Chapter 6, conclusions are drawn, followed by a discussion of directions for future work.

2

Minimizing the Impact of an Attack

In this chapter we will elaborate on how artificial diversity can help to minimize the impact of a successful attack. Software providers can use diversity to make it harder to craft a single attack which circumvents the intended behavior of all the copies of their software at once. Similarly, content providers can protect the intended use of their content by diversifying the software that regulates access to the content.

Artificial diversity can be introduced in a number of different ways. Existing applications of randomization can be found in the context of the malicious code model. The approach introduced Barrantes et al. [Barrantes 05] consists of encrypting the code at load time as a defense against binary code injection attacks. In the case of Address Space Layout Randomization (ASLR), the operating system positions key data areas in a non-constant way to make it harder to predict target addresses for the exploitation of memory-related errors.

This type of defense is less viable in the malicious host model, as we cannot rely on the environment. The only thing that we can partially control is the program itself and thus the randomization needs to be an integral part of it. Under these circumstances, there are still a number of possibilities on when and where to randomize:

- **Before distribution:** The static representation of the program is randomized before distribution.
- **During installation:** The static representation of the program is randomized when it is installed. Based upon, e.g., hardware, the license key, etc.

- **Between runs:** The static representation of the program is randomized between executions, comparable to metamorphic viruses.
- **During execution:** The dynamic execution trace of the program is randomized.

Section 2.1 discusses the advantages and disadvantages of introducing diversity before distribution and during installation, while Section 2.2 discusses randomizing the execution trace.

2.1 Break Once Break Every Time Resistance

Despite many initiatives to better protect software and content, the financial impact discussed in Section 1.1.1 is still huge. We are convinced that this is a risk that cannot be addressed sufficiently without risk diversification.

Therefore, we present an alternative technical protection scheme, whose strength is based on diversity. Diversity is an approach which is somewhat different from traditional approaches. Rather than trying to prevent or postpone every attack on the software, we accept that protection mechanisms can eventually be broken. Diversity is a business model which remains viable in the presence of cracks, because their impact is local in space and time.

In this scheme, each installed copy of a program is unique to make it harder for an attacker to generalize a successful attack on one version of the software to other versions.

Furthermore, the proposed scheme includes software updates to migrate from a static nature of defense to a more dynamic one. In particular, software updates in our scheme are crafted to ensure that they work for one, and only one, version. When updates are no longer provided for installed copies that are known to be compromised, a new line of defense needs to be broken with every critical update.

2.1.1 The Software Distribution Scheme

The core of our protection scheme consists of two levels of diversification. At the first level each distributed copy is different. At the second level every installation of a specific copy is different. We will refer to a specific copy installed on a specific machine as an *instance*, and to an instance-specific update as a *tailored update*.

An instance must be activated through interaction with the software provider and contains links to the hardware to ensure that an instance cannot simply be copied to another machine.

The software provider maintains a database that keeps track of the legitimate instances and their characteristics. The instances are crafted in such a

way that they differ significantly, allowing the creation of updates fit for one instance and one instance only in such a way that a tailored update cannot easily be generalized for other instances.

When a user requests an update, he needs to identify the instance he wants to obtain the update for. The software provider then checks if the request is legitimate, looks up the characteristics of the instance and generates a tailored update.

2.1.2 Impact on the User Base

To assess the impact of this type of protection on the user base we present an economic model. It is loosely based on the work by Conner and Rumelt [Conner 91] and Altinkemer and Guan [Altinkemer 03].

Participants

We will assume a simple content distribution model that consists of the following participants:

- **Content providers**, who want to maximize their profits now and in the future.
- **Fair users**, who are willing to pay for the content and/or use it as intended. They want to have access without being impaired by the protection mechanisms.
- **Crackers**, who have the technical skills and the desire to circumvent the protection mechanisms and want to minimize the risk of being caught.
- **Unfair users or script kiddies**, who have little or no technical skills and want to enjoy the same privileges as legitimate users without proper compensation (e.g., in the case of piracy), or who want to have their software behave as desired by the crackers (e.g., cheating in games).

One of the key observations for diversity is that the number of crackers is much smaller than the number of script kiddies. Script kiddies typically subsist on the skills of crackers who distribute license keys, patches or patched versions of software.

Consumer Behaviors

In the following discussion, we index each potential user by i. Let p be the price of the content, v_i the value of the content to the user and c_i the cost of

altering the behavior intended by the provider to the behavior intended by the user over time.

From an economical point of view, the sets of legitimate users (L), illegitimate users (I) and users who do without (D) would then be:

$$\begin{aligned} L &= \{i : p \leq \min(v_i, c_i)\}, \\ I &= \{i : c_i < \min(v_i, p)\}, \\ D &= \{i : v_i < \min(c_i, p)\}. \end{aligned}$$

Assumptions

We assume that an increase in the level of protection does not influence the price p of the content.

We furthermore assume that the perceived value of the software v_i remains constant. This implies that we ignore network externalities. Positive network externalities increase the value of content as more users (legitimately or illegitimately) use that content. In the presence of network externalities, it may be beneficial to tolerate piracy. This may lead to an increase of the exchangeability of data and an increase in the production of complimentary goods. Furthermore, it may help to lock-in consumers in an early phase. Our scheme allows for a fine-grained control over the distributed copies and as such enables piracy discrimination, as will be discussed in the next section.

Likewise, it implies that we ignore negative network externalities. An example of negative network externalities is the presence of many cheaters in an on-line game, which decreases the experience of fair users.

The Traditional Two-phase Model

Typically, software protection assumes a two-phase model. When the software is released, no cracks are available. After a certain time period, cracks can be found in public distribution channels, such as the Internet. The majority of the revenue is expected from the first period. This is a common assumption in games, where a couple of months of unbroken copy protection mechanisms are sufficient to make profit. Therefore, traditional defense mechanisms go through great lengths to increase the period of time between the release and the public availability of a crack.

The different relations between p, v_i and c_i are shown in Figure 2.1a. The public availability of cracks results in a decrease of c_i for most users.

As a result, in the second time period, a number of users who would otherwise do without or use the software legitimately, will become illegitimate. We call these users script kiddies. The transitions are indicated by the dashed arrows in Figure 2.1b.

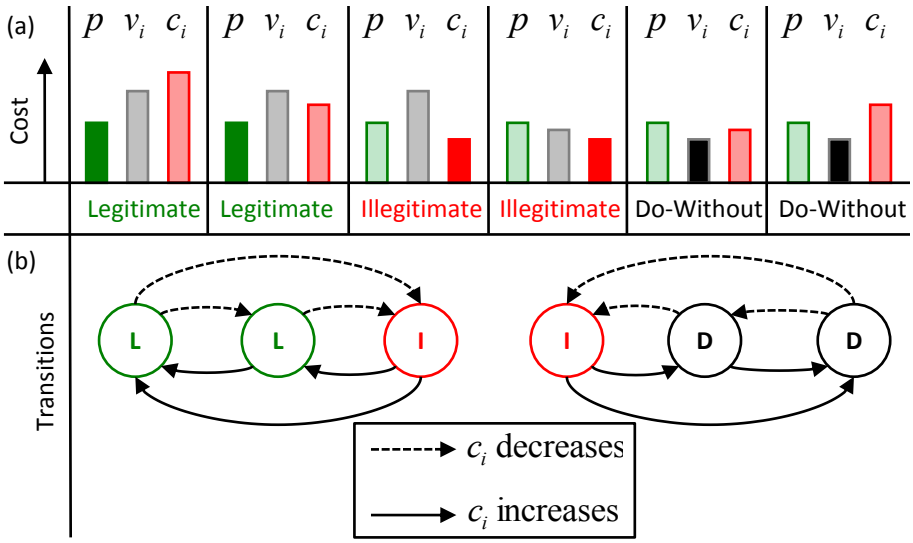


Figure 2.1: (a) Possible relations between p, v_i and c_i, (b) Possible transitions when c_i changes

Crackers are the users who find the cost of cracking smaller than the perceived value or the price when the program is first released. These users typically have skills to circumvent the protection mechanisms relatively easily.

Some users remain legitimate even when it is no longer technically difficult to be illegitimate. This is because c_i largely consists of three components:

- The perceived cost of obtaining and maintaining access to the content.
- The perceived cost of being caught multiplied by the perceived probability of being caught.
- The perceived cost of overcoming moral inhibitions (personal ethics, social factors, habit, etc.) to cracking.

A publicly available crack mainly decreases the first component.

Spatial diversity is an attempt to reduce the usability of a crack for other users. This way, the public availability of a crack for a single copy does not result in a significant reduction of c_i. As such, spatial diversity is an attempt to reduce the transition from legitimate (and do-without) users to illegitimate users (dashed arrows in Figure 2.1b). We are thus trying to block the script kiddies instead of the crackers.

Note that taking into account positive network externalities would enable transitions from people who do without to legitimate users. Conversely, negative network externalities would enable transitions from legitimate and illegitimate users to people who do without.

A Multi-phase Model

Temporal diversity tries to go from a two-phase model to a multi-phase model. By diversifying the software over time, content providers can impose new protection barriers with updates. This way, script kiddies that are illegitimate in the presence of a crack, will have to become do-without or legitimate until a cracker makes a patch publicly available for the renewed protection mechanism. Ultimately, requiring new cracks over time increases the total cost c_i of obtaining and maintaining access to the content, even for the crackers. The resulting transitions are indicated by the solid arrows in Figure 2.1b.

Cost Benefit Analysis

We denote L_d as the set of legitimate users in the presence of diversity, L_u as the set of legitimate users without diversity (with uniformity), and the cardinality of a set S as $|S|$. The application of software diversity is then beneficial if $p|L_d \setminus L_u| \geq C_d - C_u$, where C_d and C_u are the total costs in the presence, respectively absence, of diversity.

Depending on the type of diversification, the cost of the development and maintenance of the infrastructure required can be considerable. When only run-time randomizations are considered, fixed costs are limited to the development of the software for the diversification of run-time behavior.

When diversifying distributed or installed copies, updates need to be tailored as well. In this case, the marginal costs include the computing and distribution costs per instance. The computing costs include the additional cost per instance for the purchase and maintenance of hardware needed for the creation of non-identical copies and tailored updates. To reduce the computing costs, the process should be fully automated.

The pressing of CDs is no longer economically viable when all distributed copies are unique. In this case it is more advantageous to burn them, but the cost per disc will be higher. While it may be acceptable to distribute the initial software this way, physical distribution of non-identical updates on a regular basis would significantly increase the cost per instance. Fortunately, in many cases, the updates could be digitally distributed, given the widespread use of the Internet.

Furthermore, the debugging process will be complicated as error reports will be instance-dependent.

2.1.3 Practical Considerations

This section discusses a number of practical problems and possible solutions related to the proposed scheme. First, the dependency on updates is discussed, then we explore the possibilities to diversify programs and to ensure that an update only works for one instance.

Reliance on Updates The dynamic nature of the scheme is only possible through updates. In the presence of the Internet, they can be distributed easily and the updating process can be done automatically by the program. Nowadays software updates are used for the following purposes:

- to fix bugs;
- to add security patches;
- to support new hardware and new file formats;
- to keep a program compatible with other programs;
- to add new functionality.

The first four categories are considered to be critical. Updates increase the cost c_i for illegitimate users. If they want to enjoy the same privileges as legitimate users, they need to find an update suited for their instance with every update.

We believe that, for most types of commercial software, a frequency of one update every couple of months will inflict enough damage on illegitimate users to persuade them to become legitimate.

However, if the frequency of updates normally required is too low, we can artificially increase the necessity for updates. Obviously, introducing bugs or security flaws to make updates necessary is not an option and introducing new hardware is too expensive.

Software aging [Jakobsson 02] can be used for document-producing programs. In this approach, instances that are not kept up to date are unable to read the output of more recently updated instances. As such we can consider the output to be of a different file format. This decreases the value of a pirated instance as it cannot be kept compatible with legitimate instances.

Legitimate users could be favored by providing them with access to a collection of add-ons or extra features. As a result, the value of the program for legitimate users will be higher than for illegitimate users as these add-ons or features will not work for illegitimate instances.

Another approach is to move from the model in which a user pays once for the use of the software to a model where a user rents the software for a limited

amount of time. The software could disable itself, unless it is updated to enable the software for an additional time period. This is however a static form of defense, making it possible for a pirate to remove the disabling code. While the diversity ensures that each copy needs to be cracked separately, a full, cracked version could still be distributed, defeating this defense mechanism. Therefore this approach must be combined with other types of updates, which should be tailored not to work with instances where the time limitation is removed.

Diverse Instances and Tailored Updates The core of the protection scheme requires the instances to differ in such a way that updates can be tailored to work for one instance and one instance only. In this section, we will discuss a possible approach to achieve this.

A typical program consists of a large number of files, containing code or data. A distributed update contains the necessary information to convert the program to a newer version. Obviously, no information needs to be included regarding unmodified files as this would only make the update larger. Also, new files need to be included entirely.

There are two possibilities for changed files: in a full-file update the entire file is included, whereas with an incremental update only the changes over the installed version are specified. The former has the advantage that when different users have different installed versions, e.g., because some users have updated their software more regularly than others, the same update can still be provided to all the users. An incremental update has the advantage that it is smaller, but different updates are necessary for different installed versions. This problem is sometimes solved by providing updates incrementally to the original version of the files, which needs to be provided by the user by inserting the installation disk. We will now discuss how our scheme can be put into practice in both of these cases.

Full-file updates When using full-file updates it is useless to apply the diversification within a single file as the full file will be included in the update. We thus need to diversify the interfaces between the different files to ensure that a file in a tailored update cannot function correctly with an illegitimate instance.

For data files, the content can be encrypted and decrypted using an instance-specific key. The same technique can be applied to the interface between code files, in which arguments passed to functions and the values returned can be encrypted. The keys could be hidden by techniques for white-box cryptography [Chow 03] to make it more difficult to circumvent this protection.

Alternatively, all data, arguments or return values can be transformed to an instance-specific domain, provided that the computations are also transformed to this domain.

Incremental updates The main incentive for full-file updates is that it facilitates the distribution as the same update can be used by each user. Clearly this is no longer a valid argument in our protection scheme since it requires the updates to be instance-specific.

Through the use of artificial diversity, every instance can have code files that differ significantly on a binary level. When an update only specifies which bits in the existing file need to be flipped to migrate to the new version, it is clear that this will not work for other instances.

Identification of Legitimate Instances When an update is requested, the instance for which it has to be tailored needs to be identified. This can be done through a simple serial number of some sort which is assigned to an instance at activation. The database would then keep track of the serial numbers that identify legal instances. The database also contains the necessary information to reconstruct the characteristics of that instance. If the instance is considered to be legitimate, the software provider tailors an update that migrates that instance to the new version of the software.

The software provider does not need to worry about illegitimate users that request an update for a legitimate instance as it will not work for their instances. However, he will keep a log of the different requests to track illegitimate copies. When many requests for the same instance have different origins, this will rouse the suspicion of the software provider and he will classify this instance as illegitimate.

To assure that the original, legitimate owner is not damaged as a result (he may not have been aware of the piracy), he should contact him and provide him with an update that migrates his instance to a new instance, with a new serial number.

Clearly, this update should not be provided to the illegitimate users. However, if illegitimate users would choose to become legitimate through correct compensation in order to obtain full access to the updates in the future, a similar process can be used to migrate their illegitimate instance to a new legitimate instance.

We also note that a legitimate user could accidentally request an update for the wrong instance. As a result his software might no longer function correctly. Again, we do not want to damage the legitimate user that has, e.g., made a typographical error when submitting the serial number. Therefore, each update should check whether it is applied to the expected instance, e.g., by comparing a checksum over the files that will be changed to the expected checksum.

The integrity of the database needs to be assured for the correct operation of the scheme. This can be achieved by the usual means to ensure data integrity for data centers, including an off-site mirror data center.

2.1.4 Case Study: Software Piracy

Circumventing copy restrictions is probably one of the most widespread applications of tampering. In this section, we will focus on this type of tampering and discuss how the proposed scheme can defend against different forms of software piracy.

Countered Forms of Piracy

Many forms of software piracy exist and it is important that a protection scheme impedes as many of them as possible. We therefore continue with a survey of these forms, as identified by the Business Software Alliance and the Software & Information Industry Association [IPR 03, SII 00], and discuss how the presented scheme can help to make them inviable.

Cracks and Serials The first type of software piracy consists of legally obtaining an evaluation version and subsequently entering a copied license code or applying a generic patch that undoes the copy protection.

This a widespread form of piracy. It is so popular because of the small amount of information that needs to be exchanged illegally. It is clear that it is easier to illegally distribute and obtain a license code or a patch than a complete program. The problem for defenders is to find a way to increase the difficulty of enabling an additional copy by reducing the uniformity of the distributed software. In our scheme, since all distributed evaluation versions are different, a patch for one instance will not necessarily work on another instance. Moreover the distributed evaluation versions can be designed to be incompatible with the license codes of other copies.

Softlifting and Hard Disk Loading Softlifting refers to the act of piracy where one copy is legally obtained and installed on more computers than allowed.

Hard disk loading is the unauthorized installation of copies of software onto the hard disks of personal computers, often as an incentive for the end user to buy the hardware.

Both forms consist of installing software on more computers than allowed by the license. In these cases, we can expect exchange of updates between the legitimate user of a copy and the illegitimate users of the same copy. We assume that a software provider cannot easily become aware of these forms of piracy because of the limited size of the communities sharing a copy.

The proposed scheme involves interaction between the user and the software provider to activate an instance as a primary, static line of defense against

this form of piracy. Even though this protection can eventually be broken, the diversity ensures that only one copy of the software will be broken at a time.

When the first line of defense is broken, a second, dynamic line of defense is provided by the diversification at installation. As a result the characteristics of an illegitimate instance will not be considered legitimate and updates will not be provided. This way these instances cannot be kept sound and up to date. However, this diversification can be turned off as well in order to ensure that the illegal instances are identical to the legal instance. To enable the illegal instances the links between the hardware and the software need to be broken. We note that it can be harmful to change the installed code to remove the links to the hardware as it might cause subsequent updates to fail. Changing the program might change the characteristics which determine the update. An alternative attack would be to emulate the hardware.

Both lines of defense need to be broken before illegitimate users can fully enjoy the software. As a result of the diversification an attack would ideally work on only one distributed copy. As each copy needs to be cracked separately and as we assume a small number of pirates, we can conclude that, given the limited size of the sharing communities, this form of piracy will no longer have a significant impact on the revenues of the software provider.

Internet Piracy and Software Counterfeiting Internet piracy is the act of making unauthorized copies of copyrighted software available to others electronically. Software counterfeiting is the illegal duplication and distribution of copyrighted software in a form designed to make it appear legitimate.

Both forms consist of installing software on more computers than allowed. The scale is typically larger than of the forms discussed in the previous paragraph. We assume that there cannot be a continuing interaction between the pirate and the illegitimate users, as the exposure of the pirate and thus the risk of legal action against him would be too high.

We only consider the case in which both lines of defense are already broken. We limit ourselves to this case because of the following reasons: if the first line of defense is not broken and the user is charged per activation of an instance, this will bring no harm to the software provider. It will instead be a free distribution and advertising channel. If the second line of defense is not broken illegitimate users will not be able to keep their software sound and up to date, since we ensure that the update for a specific instance cannot easily be derived from an update targeted towards another instance.

In practice, the majority of versions pirated this way has the same origin. For example, most of the pirated versions of Windows XP were tied to a few volume license product keys [Mic 02]. Given the large scale, the software provider probably can become aware of the piracy, for example, by searching the Internet for illegally distributed copies. Alternatively, as these illegitimate

versions need to be kept sound and up to date and there cannot be a continuing interaction with the original pirate, many requests for updates will be made from many different locations for the same instance. This would also rouse the suspicion of the software provider. If an instance is considered to be corrupted, the software provider will no longer provide updates for these instances, thereby impairing the illegitimate users.

Mischanneling Mischanneling refers to the form of piracy where, e.g., an academic license is used for commercial purposes. To our knowledge this is the only form of piracy of an entire program against which our scheme does not provide protection. In fact, we are not aware of any technical protection against this form of piracy.

Piracy Discrimination

As discussed, piracy can help to lock in consumers in an earlier phase and can lead to higher profits in a later phase. Furthermore, software piracy can create additional positive network externalities. In our model, software providers want to maximize their profits now, but also in the future. Users that have illegitimately used a program for a while and want to turn to a legal version can be expected to stick to the software they are used to because switching implies additional costs. These include learning costs, transaction costs (because the installation and implementation of a new software system is not a trivial task) and artificial costs: an update can for example be cheaper than the full version. In some cases the pirate does not have the financial capabilities of obtaining the software legally, but he can be expected to do so in the future.

The proposed scheme enables a fine-grained control over the distributed copies and allows a software provider to tolerate an arbitrary level of piracy. This is made possible through the activation and the tailored updates. A software provider can choose which instances are activated and for which instances updates are provided. Furthermore he knows the origin of the instance for which an activation or update is requested because of the diversity.

For example, two installations of a copy for private use could be allowed to enable a user to use the same copy on a desktop and on a laptop. On the other hand, a copy for commercial use can be limited to one installation or exactly as many as agreed upon in the license.

2.2 Break Once Run Every Time Resistance

The diversity discussed in the previous section is applied before distribution. In this section, we propose to combine the best of both worlds — the benefits

of diversity and near-zero marginal costs — by introducing diversity after the distribution.

The ideas are discussed in the light of a specific model of a tamperer's behavior: the locate-alter-test cycle. It has been understood for long that there are many similarities between tampering and debugging. In this model, we make these similarities explicit. As a result, the techniques presented to counter tampering leverage known difficulties from the domain of debugging: indeterministic behavior (from the viewpoint of the program) and the fundamental limitations of testing (for every input, every environment). Despite originating from a specific model, the techniques increase the workload to create a fully functional patched version in a more general attack model, which assumes only that behavioral changes are made by modifying the program itself.

The underlying ideas are that (i) An attacker typically repeats the execution of the program with a particular input and slowly zooms in on the part where he thinks a vulnerability may occur. This becomes harder if the execution cannot be replayed at will and (ii) If we can fool an attacker into believing that he has succeeded for a longer period of time, we can delay the feedback loop of software tampering.

2.2.1 Low-level Debugging versus Tampering

Debugging and tampering are similar in many respects: many of the same techniques and tools are used in both disciplines. Debugging is about finding and reducing the number of defects in a computer program to make it behave as the software provider intends. Likewise, tampering is about finding and reducing the number of undesired features to make it behave as the user desires.

The incentive to tamper with software thus originates from the difference between the behavior intended by the software provider and the behavior desired by the user. Examples have been given in Section 1.1.1.

Put another way, debugging is about transforming the semantics encoded in the program to the semantics intended by the software provider. Tampering is about transforming the semantics encoded in the program to the semantics desired by the user. Therefore, it should be no surprise that both disciplines are alike. Many tools, such as IDAPro and SoftICE, and many techniques, such as breakpoints and slicing, have been originally designed for debugging, but are heavily used in tampering. The main difference is that during debugging a higher-level representation of the program is often available (such as source code or specification), while tampering typically starts from machine code or bytecode.

Similar to the edit-compile-test cycle of debugging, tampering is typically a cyclical process. Since tampering is usually done at a low level, the compile

phase can be eliminated. Furthermore we can split up the edit phase, leading to the following cycle:

1. **Locate the origin:** To turn the observed undesired behavior into desired behavior, a tamperer first needs to find the origin of the undesired behavior. For example, the displayed health of a game character is only a manifestation of the internal state. Locally changing the code that displays his health will not result in the desired behavior. He needs to trace it back to where the internal representation of his health actually gets decreased.

2. **Alter the behavior:** Once the origin is determined, a tamperer needs to determine and apply a set of changes that will alter the undesired behavior into desired behavior.

3. **Test:** In this phase, the tamperer checks whether the behavior of the software is as desired. If so, his work is done. Otherwise, more cycles are required.

2.2.2 Slowing Down the Locate-Alter-Test Cycle

If tampering is similar to debugging, we can argue along the same lines that making tampering harder is the opposite of making debugging easier.

One of the key concepts in making software easier to debug and maintain is modular design. Modular design facilitates local changes and thus minimizes the need to verify the impact of a local change on other parts of the program. Most tamper resistance techniques [Chang 02, Chen 02, Horne 02] have focused on doing the opposite: making the program more inter-dependent. Existing techniques are thus about **slowing down the alter phase** by requiring an understanding of a larger portion of the program and more binary changes to possibly unrelated sections of the program to effect a small change in the behavior of the program.

In this line of work, we focus on slowing down the locate and test phase.

Slowing Down the Locate Phase

Looking again at debugging, the first task when dealing with a bug report is to reproduce the problem. This is vital, since one cannot observe a problem and learn new facts if one cannot reproduce it. Furthermore, reproduction is essential to find out whether the problem is actually fixed. Reproducing is one of the toughest problems in debugging. One must recreate the environment and the steps that led to the problem [Zeller 05].

Similarly, reproducing undesired behavior is indispensable for tampering. The manifestation of undesired behavior needs to be traced back to its origin. Typically, a tamperer repeatedly executes the application with a particular input and slowly zooms in on a part where he thinks the undesired behavior may originate.

This requires that execution can be replayed at will. We try to hamper this process by choosing between different control paths based on pseudorandom numbers, timing results, thread scheduling, etc.

In software tamper resistance, the "bugs" are features which we want to manifest every time, so it seems illogical to make their appearance indeterministic. We can, however, make sure that these features manifest themselves in different ways by duplicating parts of the program, diversifying them and choosing more or less randomly among the alternatives at run time. This makes it harder for a tamperer to zoom in on the vulnerable part of the program, as the semantics of the program may be constant, but the execution paths will not be identical.

Slowing Down the Test Phase

Testing is also a major issue in debugging and software maintenance. It is very hard to foresee every input, every environment, every usage scenario and every combination of applications [DiMarzio 07]. Testing can only show the presence of undesired behavior, not its absence.

The techniques discussed in this section increase the number of tests required to manifest all occurrences of the undesired behavior. The underlying idea is that the impact of a successful patch for a small subset of the input space, for a small number of computers or for a short period of time does not pose a great threat to the software or content provider.

The time required to create a fully functional tampered version of the software is increased by letting the tamperer believe that he has succeeded, while the attack only works for a subset of the input space, or for a short period of time. Tamperers often work by trial and error. Using incomplete knowledge about the program of which they change parts, hoping that the desired results will arise. When it is easy to evaluate whether these results have been obtained, this process can be repeated many times. If this evaluation takes longer, e.g., because it works for most of the input sets most of the time, the workload increases.

Furthermore, the credibility of the tamperer in the cracker community may decrease if he claims to have successfully patched a program, while it still behaves as intended by the software provider on other computers.

We could for example use one type of license check in 90% of the cases and another one in the remainder. This way, the tamperer may be fooled into

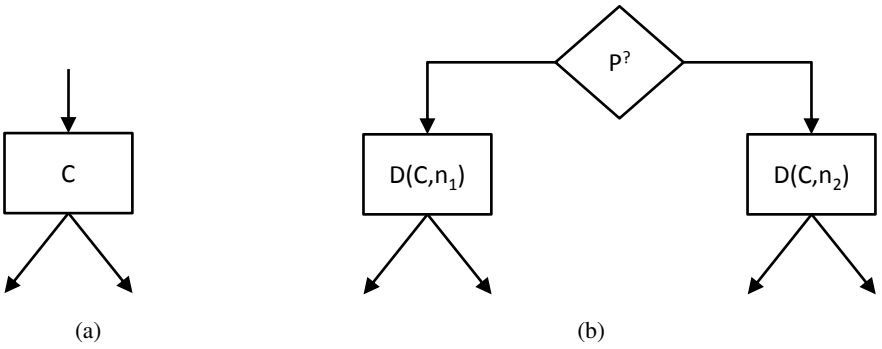


Figure 2.2: The basic mechanism behind run-time randomization

believing that he has succeeded for a longer period of time. In this case, the tamperer has done a good job if the undesired behavior appears randomly: he can just restart the program and hope that it will work next time. However, if it is linked to certain input patterns or hardware identifiers, the usability of the tampered version is decreased significantly.

2.2.3 Tools of the Trade

The core mechanism behind the discussed techniques is illustrated in Figure 2.2. In its simplest incarnation, a fragment of code is duplicated, both copies are diversified and, at run time, one of them is selected more or less randomly through a two-way opaque predicate.

An opaque predicate is a predicate that has a property that is known at obfuscation time, but for which this property is hard to prove afterward. A two-way opaque predicate ($P^?$) has the property that its outcome does not affect program behavior. This section goes into more detail on two aspects related to the input of the two-way opaque predicates. Firstly, we explore techniques to augment the user-observed input with chaff input as a source of randomness. Secondly, we will discuss the usage of variable program state at a program point, e.g., as fake input dependencies. The generation of diverse copies is discussed in Chapter 4.

Chaff Input

We say that run-time randomization delays the locate phase if it introduces diversity during a single "tamper session", i.e., if the randomization takes place even on a single computer, for the same user-observed input and for a limited period of time (one or a few days). Conversely, run-time randomization is said

to delay the test phase if it requires multiple tamper sessions, i.e., the tampering itself needs to be repeated for different computers, for different user-observed inputs, for different periods of time.

Under these specifications, chaff input is needed to delay the locate phase. This will be used as a source of randomness, which will then serve as input for the opaque predicates. Note that chaff input is likely to stand out in command-line applications, as there is typically little difference between the user-observed and the fully specified input. However, many interactive applications already make use of threads, timing information, information about mouse movement, on-line content, etc. making them more suited for this technique.

We will now discuss some sources of chaff input.

- **Scheduling of threads** In multi-threaded applications several threads may interact with each other in a non-deterministic manner (from the viewpoint of the application). The actual scheduling depends on the operating system (and the virtual machine, if applicable), and is influenced by asynchronous events such as user interaction, other processes, thread priority, and so on. Therefore, the actual scheduling is an excellent source of randomness. If necessary, additional threads can be created to perform part of the original functionality, or to perform other software protection tasks.

- **Return values of system calls** System calls also provide a source of randomness from the viewpoint of the application. Many system (or library) calls return information that is changeable over different runs: system time, unallocated memory, network traffic, load of the machine, file system, etc.

  The underhanded C code contest 2005[1] contains examples on how to obtain randomness in a covert way. One of the entries leaves a matrix partially uninitialized, as a result of which it still contains information from a previous `stat()`-call (`stat` returns file info, including time of last access). This type of call is common in regular programs and will thus not quickly raise suspicion.

  An interesting way of randomizing the program is to change the code executed (not the behavior) based on the presence of a debugger. This way an attacker could spend a lot of time making the program behave as desired in the debugger, only to find that it behaves differently without the debugger.

[1] http://www.brainhz.com/underhanded/

- **Side effects of user input** A user generates more input than typically used by the program. For example, the time between different key-strokes is unlikely to affect program behavior. Similarly, when a button is clicked, the exact mouse coordinates are often ignored. Furthermore, the path along which the user has reached the button is often not taken into consideration. By including this type of information to guide the execution, we can add more diversity.

- **External service** Alternatively, we may require access to an external service, which provides a source of randomness. Such an external service could be trusted hardware or an on-line service.

Record/Replay Mechanisms and Omniscient Debuggers

Clearly, given a fully specified input, the behavior will be deterministic. While the fully specified input is often a superset of what the user perceives as input, a tamperer could ultimately use a perfect record/replay system [Ronsse 99] to make the fully specified input (including data, user interaction, communication, system calls and scheduling events [Zeller 05]) repeatable, and thus the execution repeatable. This way he can track down and tamper with one of the copies of the origins of the undesired behavior. Alternatively, he could use an omniscient debugger [Bhansali 06], e.g., the Simics Hindsight Debugger, to backtrack to the origin. Note that the general application of these techniques can be very expensive in terms of memory requirements. Therefore, a potential defense against such capabilities is to increase the amount of state necessary for the debugger to be able to trace backwards. This can be accomplished by maximizing the number of irreversible operations in the program.

In any case, there will be more origins of the undesired behavior. Unless the tamperer finds a way to automate detecting copies of that specific origin, which is undecidable in general, he has to either (i) repeat this labor-intensive method for every copy of the origin, or (ii) he has to make the choice between the different copies fixed. As a result, the workload on the tamperer increases.

Fixing the choice between different copies may be complicated as well. It may be easy to automatically find points where the different executions digress, but some of these points may be part of the original functionality of the program.

Variable Program State and Fake Input Dependencies

The internal state at a program point is itself highly variable and, therefore, an excellent source of input for the two-way opaque predicates. Furthermore, it is less suspicious to select different execution paths based upon the internal state.

Through profiling [Graham 82, Pettis 90], we can easily spot tuples (p, s), for which either (i) the state s is constant at program point p for a fixed input, but variable for different inputs, or (ii) the state s is variable at program point p even for a fixed input. Note that, due to the nature of profiling, we cannot be certain that a state s is fixed, we can only conclude that a state s is fixed for the tested inputs.

Tuples for which the first property holds are candidates for introducing fake input dependencies. As a result, the execution for different inputs will differ at places where it originally overlapped, thereby delaying the test phase.

Tuples for which the second property holds are useful to delay the locate phase, because they will increase the amount of information in the static representation of the program and the number of different instructions in a trace of a particular execution. As a result, the trace will be less "foldable", by which we mean that constructing a Control Flow Graph (CFG) from the trace will result in a larger CFG than from the original program.

Using the same argumentation as earlier, an attacker needs to patch at least one of the copies, and needs to patch additional ones or remove the fake dependencies on the program state.

3
Matching System – Menelaus

A matching system tries to identify related code fragments between two programs. As such, it can be used to help identify similarities and differences between programs. There are numerous situations in which this is useful:

- During an incident response, one may find a modified version of some program and want to know which modifications have been made to it.
- An undocumented third-party Application Programming Interface (API) may have changed and you need to figure out what has changed in a new version or after a patch.
- Crackers may want to use the difference between a version before and after patching to figure out how to set up an attack against unpatched systems.
- System administrators may want to verify the impact of a patch or check if a bug has indeed been fixed.
- Crackers may have crafted an attack against a single version and try to generalize it to other versions. For example, a conditional branch guarding the outcome of a license check may be a single point of failure for a copy protection mechanism. A cracker can save time breaking another version if matching techniques can readily indicate the corresponding conditional branch in the other version.
- It can help to prove that some code fragment has been pirated.

- It can facilitate detecting the modification responsible for a bug (delta debugging).
- It can be used to port information between different versions. Profile information, for example, is expensive to collect and has been shown to be portable to newer versions [Wang 00].

This chapter discusses a matching system that can be useful in all of these applications. However, our main goal is to evaluate how successful related code fragments can be matched between different versions.

3.1 Quality of a Matching System

We informally define a matching system for code as follows. Let C be the set of all syntactically correct code fragments. In this dissertation, a code fragment is a sequence of one or more assembly instructions starting at a particular address. Two granularities will be considered: single instructions and basic blocks.

For notational simplicity, we will consider a program to be a set of code fragments. A matching system M takes as input two programs $A, B \in \mathcal{P}(C)$ and produces a mapping μ. A mapping is defined as a subset of the pairs of code fragments from A and B ($\mu \subseteq A \times B$). M is thus of the following type:

$$M : \mathcal{P}(C) \times \mathcal{P}(C) \mapsto \mathcal{P}(C \times C) \ .$$

We will refer to the output of a matching system M for two programs A and B as the estimated mapping: μ_e. To determine the correctness of this estimated mapping, we need a reference mapping μ_r, which defines which code fragments are related (belongs to μ_r) and which are not (belongs to $\overline{\mu_r}, \overline{\mu_r} = (A \times B) \backslash \mu_r$). These sets are illustrated in Figure 3.1. For now, we will simply assume that such a mapping exists. We will discuss in the next chapter how such a mapping can be obtained in practice.

The difference between the estimated mapping μ_e and reference mapping μ_r is an indication of how well we have succeeded in matching code fragments between the two versions. As usual, there can be two types of errors in the estimated mapping: false negatives and false positives.

3.1.1 False Negatives

False negatives are pairs of code fragments in the reference mapping that are not in the estimated mapping, i.e., the set of false negatives is $\mu_r \backslash \mu_e$.

A lower number of false negatives indicates a better estimate. Intuitively, this indicates how well the matching system can link related code fragments.

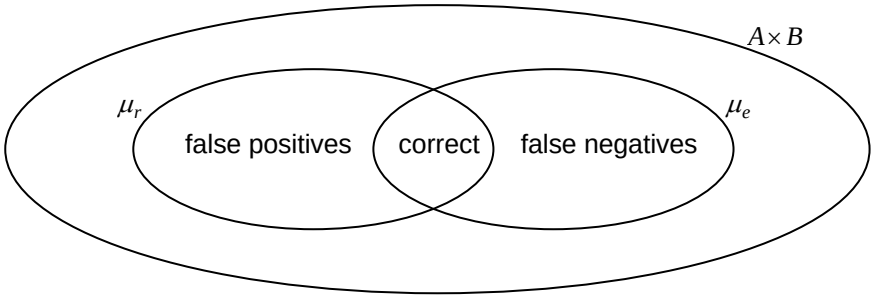


Figure 3.1: The reference and estimated mapping are subsets of AxB

If a pair of related code fragments is in the estimated mapping, those code fragments are also correctly perceived as related by the matching system. Otherwise, they are falsely perceived as unrelated.

For evaluation purposes, we will use the *false negative rate* (ν): the fraction of pairs of code fragments in the reference mapping that is not in the estimated mapping, i.e.,

$$\nu = \frac{|\mu_r \backslash \mu_e|}{|\mu_r|} \quad .$$

A false negative rate of 0.5, for example, means that we have only found half of the required matches. Clearly, it is trivial to reduce the false negative rate to zero by including every pair of fragments in the estimated mapping ($\mu_e = A \times B$). However, this will result in many reported pairs of unrelated code fragments.

3.1.2 False Positives

False positives are pairs of code fragments that are not in the reference mapping, but that are in the estimated mapping, i.e., the set of false positives is $\mu_e \backslash \mu_r$.

A lower number of false positives indicates a better estimate. Intuitively, this indicates how well the matching system can *distinguish* unrelated code fragments between two versions. If a pair of unrelated code fragments is in the estimated mapping, they are incorrectly perceived as related by the matching system. Otherwise, they are correctly perceived as unrelated.

For evaluation purposes, we will use the *false positive rate* (π): the fraction of pairs of code fragments in the estimated mapping that are not in the reference mapping, i.e.,

$$\pi = \frac{|\mu_e \backslash \mu_r|}{|\mu_e|} \quad .$$

A false positive rate of 0.5, for example, means that for every correctly reported match, we have one falsely reported match.

In some cases, we will also indicate the fraction of the worst case of false positives that is reported in the estimated mapping (ϕ), i.e.,

$$\phi = \frac{|\mu_e \backslash \mu_r|}{|\overline{\mu_r}|} \quad .$$

Clearly, it is trivial to reduce the false positive rate to zero by not including any pair of code fragments in the estimated mapping ($\mu_e = \emptyset$). However, this will result in many unreported pairs of related code fragments.

Therefore, we will quantify the quality of a mapping by a pair of numbers: the false negative rate and the false positive rate.

3.2 Fuzzy Classifiers

Our prototype matching system is built on top of a number of classifiers. A classifier c will help to estimate whether or not a pair of code fragments $(a, b), a \in A, b \in B$ is related. An ideal classifier c would return 1 for a pair of code fragments that is supposed to match, and 0 otherwise, i.e.,

$$c(a,b) = \begin{cases} 0, (a,b) \in \overline{\mu_r} & , \\ 1, (a,b) \in \mu_r & . \end{cases}$$

In general, such an ideal classifier is impossible to create. Therefore, we settle for fuzzy classifiers. Each of these fuzzy classifiers will indicate its confidence as to whether or not to match two instructions by returning a value in the interval $[0, 1]$.

Alternatively, the classifier may have no information available on the code fragments in question. In that case, it can return the "no opinion" value: $\perp$. This value will be used in situations where the classifier has no information available for the code fragments under consideration. As we will see later, it will behave as the value 0 when selecting new pairs for the estimated mapping μ_e, and as the value 1 when removing pairs from the estimated mapping. This will ensure that the status ($\in$ or $\notin$ μ_e) of a pair for which no information is available does not change.

Precise Information

Conceptually, the input of a classifier is made up of two code fragments, as described above. In practice, the code fragments will be augmented with different types of information, depending on the classifier at hand.

If the software has been diversified deliberately, we cannot rely on debug information or heuristics about the compiler or linker used to obtain this information. Furthermore, it can be expected that measures will have been taken to prevent disassembly and control flow graph construction.

Therefore, we rely on a dynamic instrumentation framework to collect information about the versions: DIOTA [Maebe 02]. The advantage of such a framework is that it does not require reverse engineering, program understanding tools or heuristics about the compiler or linker used. The basic idea is that instrumented code is generated on the fly, while the original process is used for data accesses. As such, the framework deals correctly with traditionally hard to instrument features such as data in code and code in data, as there is no uncertainty about the code that actually gets executed. Therefore, this information is guaranteed to be correct and accurate [Ernst 03].

3.3 Experimental Setup

The strength of a matching system will be evaluated with respect to a diversity system. In this chapter, we will use a trivial diversity system consisting of a single transformation: the identity function. The reference mapping μ_r is trivial to obtain for this diversity system: code fragments are related to, and only to, the code fragments at the same address in the other version. A non-trivial diversity system will be introduced in the next chapter.

Evaluating a matching system on two identical programs can already provide useful insight in how well a matching system can recognize unrelated code fragments.

False Positive Rate

We will use two types of graphs in this chapter to evaluate the different classifiers. The first type of graph plots the false positive rates π and ϕ for both levels of granularity: basic block level (bbl) and instruction level (ins). See, for example, Figure 3.2(a).

The x-axis denotes different thresholds for the confidence of the classifier. The set of estimated mappings μ_e is obtained by including every pair of code fragments for which the classifier returns a confidence higher than the threshold. The y-axis indicates the false positive rate.

For example, the 1950 related pairs of basic blocks, which make up the reference mapping, are all assigned a confidence of 1 with respect to the classifier at hand. Furthermore, 7910 pairs of unrelated basic blocks result in a confidence of 0.98 or more. If we were to include every pair with a confidence of 0.98 or more, we would have 7910 false positives on a total of 9860

($= 7910 + 1950$) reported matches, or a false positive rate of about 0.8. Hence the value of 0.8 on the y-axis for x=0.98 for π(bbl).

Equivalence Classes

The second type of graph shows the number of equivalence classes (along the y-axis) of a given size (along the x-axis). Note that, in the interest of clarity, we do not show the number of equivalence classes of a given size if the number is zero. The numbers in the graph deal with the static portion of the code that is executed at least once.

Equivalence classes are defined by an equivalence relation. We can derive such a relation from the fuzzy classifier c as follows:

$$\forall (a, b) \in C^2 : a \approx b \Leftrightarrow c(a, b) = 1 \ .$$

To make this relation an equivalence relation, we impose the following restrictions on the classifiers:

$$\forall a \in C : c(a, a) = 1 \ , \qquad \text{(Reflexivity)}$$
$$\forall a, b \in C : c(a, b) = 1 \Rightarrow c(b, a) = 1 \ , \qquad \text{(Symmetry)}$$
$$\forall a, b, c \in C : c(a, b) = 1 \wedge c(b, c) = 1 \Rightarrow c(a, c) = 1 \ . \qquad \text{(Transitivity)}$$

As a result of the first requirement, the false negative rate ν is 0 for any threshold of confidence when considering the trivial diversity system.

Benchmark

The evaluation in this chapter considers the mcf benchmark from the SPEC CPU2006 benchmark suite. This benchmark uses combinatorial optimization to solve a single-depot vehicle scheduling problem. It has been compiled with gcc 3.2.2 and statically linked to glibc 2.3.2. The tracing of the benchmark has been done with the test input set. During this run, 1950 different basic blocks and 8494 different instructions are executed.

3.4 Classifiers Based on Local Information

In this section, we will discuss a number of classifiers which operate on additional information about the code fragments under consideration. In the next section, we will take into account the proximity of the code fragments to other code fragments.

The building block of the code of a program is the instruction. Therefore, we will first use this granularity for code fragments. As will become apparent when evaluating the quality of the classifiers, many properties of instructions are not sufficiently discriminative to distinguish similar (or identical), but unrelated instructions.

This comes as no surprise: just consider the number of times instructions such as the jump instruction and the return instruction appear in a program at different locations. In order to partially mitigate this problem, we can take into account a broader context of the instruction, the basic block. By operating at the more coarse-grained basic block level, we will leverage the knowledge that the instructions in the basic block are always executed in combination. This will help to decrease the false positive rate.

3.4.1 Instruction Syntax

One of the most obvious types of information to collect is the syntax of the instructions that make up the program. This information corresponds to the information that would be available from an assembly representation of the instructions that have been executed during a particular run. This includes the opcode, source and destination operands.

The classifier reflects how comparable the opcode and operands are between two instructions.

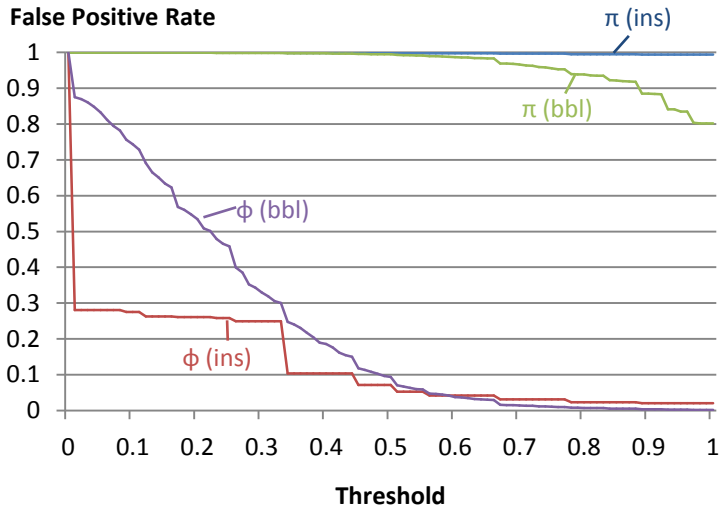
Evaluation

The graphs in Figure 3.2 evaluate the discriminative strength when applied to the trivial diversity system. As can be seen from part (a) of the figure, the false positive rate at the instruction level is very high. This means that a lot of pairs of instructions will be considered related wrongfully.

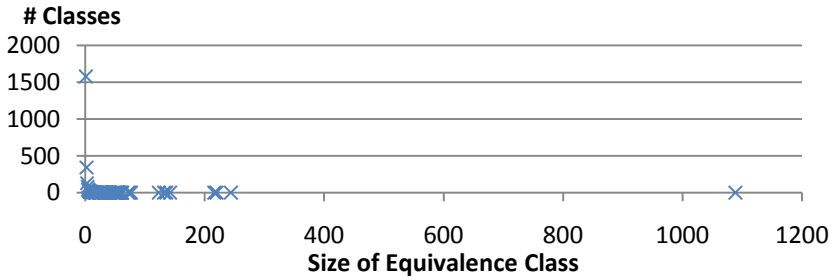
Even at threshold 1, were we include only pairs of instructions that appear to be identical in the estimated mapping, we still have a very high number of false positives. A more in depth analysis of this phenomenon is given in part (b), where we have plotted the number of equivalence classes (y-axis) of size n (x-axis) at the instruction level. For example, there is one equivalence class of size 1089. As these instructions cannot be distinguished based on instruction syntax, they will all be considered to be related, resulting in $1089 \times 1089 - 1089$ false positives. This results from each instruction having exactly one related instruction. As such, the false positives generated by this equivalence class alone dwarf the true matches (8494).

Manual inspection shows us that the most frequently appearing instructions are:

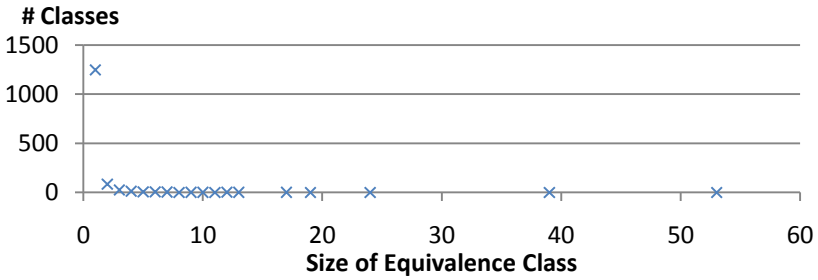
`jcc`	1089 times
`testl %eax %eax`	244 times
`jmp`	220 times
`call`	216 times
`ret`	142 times



(a) False positive rates



(b) Equivalence classes at instruction level



(c) Equivalence classes at basic block level

Figure 3.2: Evaluation of the classifier based on instruction syntax

The reason we cannot discriminate between different (conditional) jumps, calls, etc. is that we do not take into account encoded offsets and addresses. This measure has been taken in order not to be fooled by reordering transformations. Clearly, for the identity diversity system, taking this information into account would yield better results.

For the mcf benchmark, 18.5% of the instructions can be discriminated perfectly. These instructions are mainly exotic instructions and instructions with encoded rarely-occurring constants.

For the basic block granularity, we can observe slightly better results in part (a). Still, at threshold 1, about 8 in 10 reported matches are false positives. Again, a more in depth analysis is given in part (c). The most frequently appearing basic block consists of a single jump instruction. Other frequently appearing basic blocks are testing the value of a register for zero and branching conditionally upon the outcome of the test.

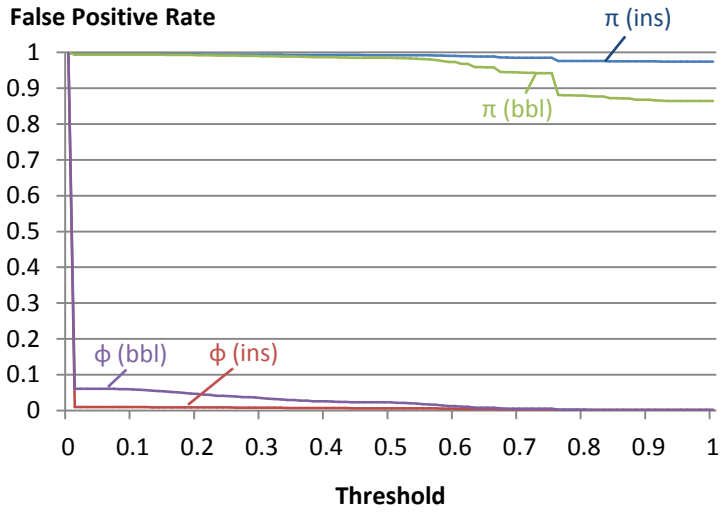
About 65% of the basic blocks can be discriminated perfectly. As a result, if we would only match basic blocks in an equivalence class of size 1, we would have a false negative rate of 35% and a false positive rate of 0%. This is an example of how limiting the number of matches per code fragment can substantially improve the results of a matching system. In practice, there are typically only a couple of correct matches per code fragment. We will use this assumption again later on to decrease the false positive rate.

3.4.2 Data

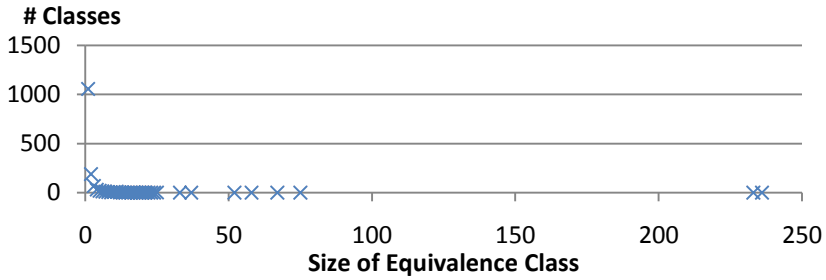
We collect the values read and written by the instructions. As this may become too much information to store, we limit this collection to the first n executions of each instruction, where n can be chosen depending on the (expected) size of the trace.

The classifier simply counts the number of matching values read and written by the instructions under consideration, independently of the location of the data, to avoid being fooled by data reordering or register reassignment. Furthermore, we exclude data values that look like addresses and the values in the set $\{-1, 0, 1\}$. The latter are excluded because they occur so frequently that they would result in many false positives.

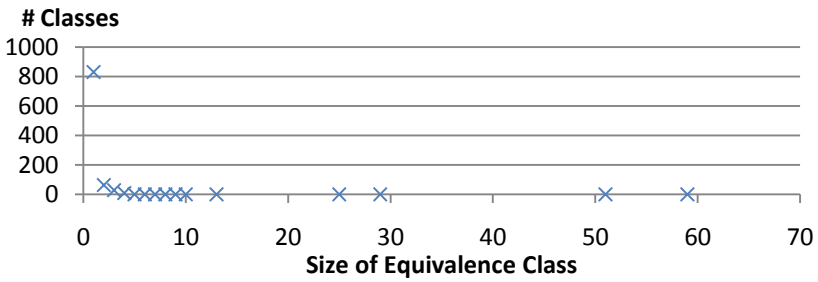
In some cases (e.g., an unconditional jump), no data is available. When the classifier has to deal with two code fragments which both have no data available, the classifier will report that it has no opinion on whether or not the two code fragments should be matched. If only one of a pair of code fragments has no data available, it will return 0.



(a) False positive rates



(b) Equivalence classes at instruction level



(c) Equivalence classes at basic block level

Figure 3.3: Evaluation of the classifier based on data

Evaluation

Similarly to the evaluation of the classifier based on instruction syntax, the graphs in Figure 3.3 evaluate the discriminative strength of the classifier based on data when applied to the trivial diversity system. For the purpose of evaluation we have considered the first five executions of each code fragment ($n = 5$).

The false positive rate is very high for this classifier. Furthermore, the classifier has no opinion on a large number of basic blocks (32%) and instructions (57%). When looking at the data values responsible for the larger equivalence classes, we observed low integer values (e.g. 2,3,4,8,10) and flag values (0x1000, 0x10).

3.4.3 Execution Count

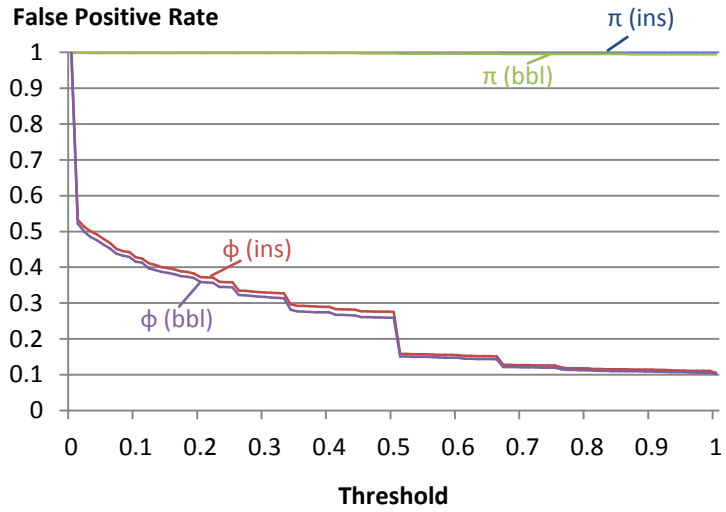
We keep track of the number of times each instruction is executed. We believe that the order of magnitude of the execution count is hard to change. It is possible to manipulate execution counts slightly, for example, decreasing them by loop unrolling or duplication and increasing them by executing code from contexts in which it only produces dead values. However, the order of magnitude is harder to change: if an operation needs to be performed a million times, it is no longer feasible to keep unrolling it until each instance of that operation executes only, e.g., ten times. Furthermore, increasing the execution count of less frequently executed or garbage code can have adverse effects on the performance of the application and is therefore best avoided.

If $\mathrm{freq}(a_i)$ is the number of times a code fragment a_i has been executed, then the score of the classifier C for a pair of code fragments (a_i, b_j) is determined by the following formula:

$$C(a_i, b_j) = 1 - \frac{|\,\mathrm{freq}(a_i) - \mathrm{freq}(b_j)|}{\max(\mathrm{freq}(a_i), \mathrm{freq}(b_j))} \quad .$$

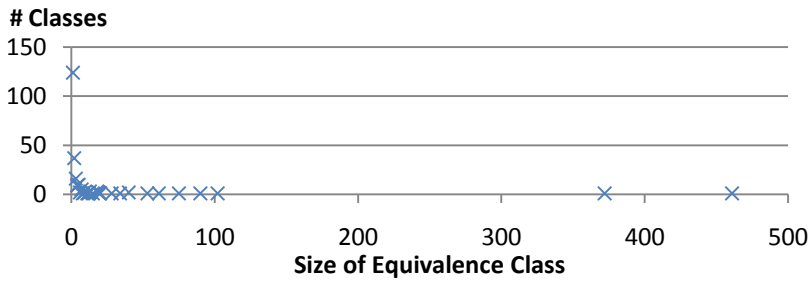
Evaluation

The evaluation of the classifier based on execution count is shown in Figure 3.4. The top five of most frequently occurring execution counts for basic blocks is as follows: 1 (461), 2 (372), 3 (102), 265319 (90), 4 (75). This is more or less what could be expected: a lot of basic blocks are infrequently executed. As such, they will create a lot of false positives. Apparently, 90 basic blocks have been executed 265319 times, which indicates a relatively large loop.



(a) False positive rates

(b) Equivalence classes at instruction level



(c) Equivalence classes at basic block level

Figure 3.4: Evaluation of the classifier based on execution count

3.4.4 System Calls

System calls are a very useful source of information about the program. All information passed to the operating system needs to be converted back into a format readable by the operating system (assuming the operating system itself is not part of the diversification). In many cases, this means that the data needs to be normalized, requiring a normalization routine to be present in the program. Furthermore, it is hard to remove or reorder existing system calls or to execute redundant system calls in a non-trivial way.

The same arguments hold for library functions if they are not part of the diversification. In this work, we only consider statically linked programs and assume that the libraries have been included in the diversification. Therefore, this is not discussed any further.

We keep track of the system calls and the information passed to them. The associated classifier scores the system calls based upon the correspondence between the values passed.

Evaluation

The classifier based on system calls yields perfect results: no false positives and no false negatives (except at threshold 0). Unfortunately, only 14 system calls appear in the program. As a result, this classifier has no opinion on many instructions and basic blocks.

3.4.5 First Execution Time

Finally, we keep track of the first time instructions are executed.

The classifier measures the difference between the actual first time it was executed, compared to the expected time. The expected time is computed by taking the difference between the total number of executed instructions of each version into account. This is illustrated in Figure 3.5. If, for example, version 1 has five executed instructions and version 2 has ten, then the expected time of the counterpart of instruction ins_v1.3 is: $3 \times \frac{10}{5} = 6$. If our classifier would then score the pair ins_v1.3 and ins_v2.10, the score would be: $\frac{|10-6|}{10}$.

Evaluation

The evaluation of the classifier based on first execution time is shown in Figure 3.6. The classifier behaves very similarly at both granularities. At the instruction level however, the false positive rate is not zero at threshold 1. The origin of that behavior is that we assign the same first execution time to every instruction in a basic block. As a result, the graph of equivalence classes in part (b) indicates how many instructions there are in basic blocks of the sizes

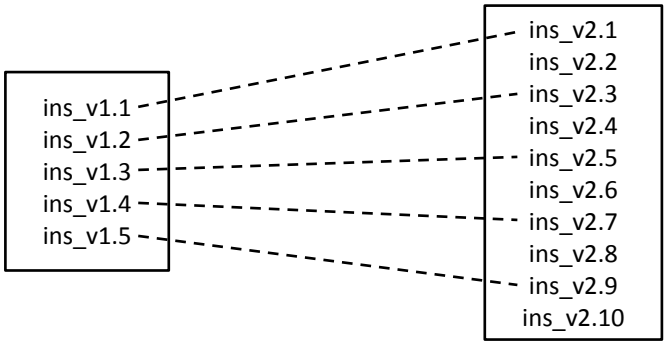


Figure 3.5: Expected execution time takes into account the difference in total size between the versions. The dashed lines connect instructions that get a perfect score according to the classifier based on first execution time.

given by the size of the equivalence classes. For illustrative purposes, we have also plotted the number of instructions divided by the sizes of the equivalence classes, as this indicates the number of basic blocks of a given size.

3.5 Proximity-based Classifiers

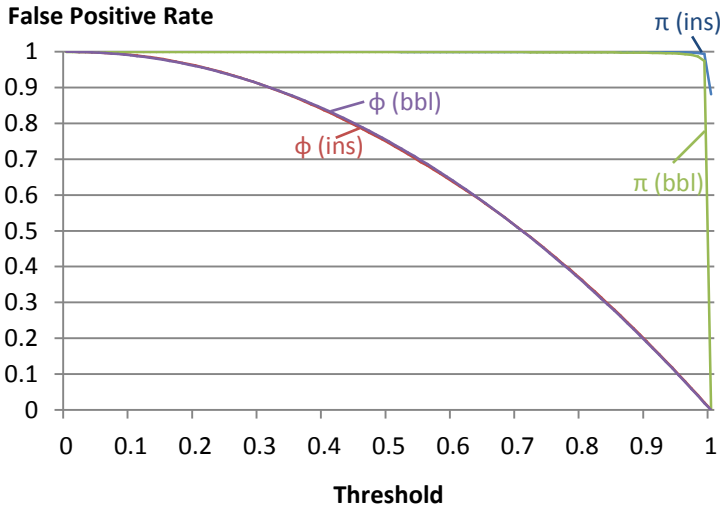
The previously discussed classifiers are based upon information local to the code fragments under consideration. The evaluation of the false positive rates for basic blocks and instructions shows us that looking at a larger context leads to better false positive rates.

By taking into account neighboring blocks in the control flow or data flow graph, we can take into account even more context information.

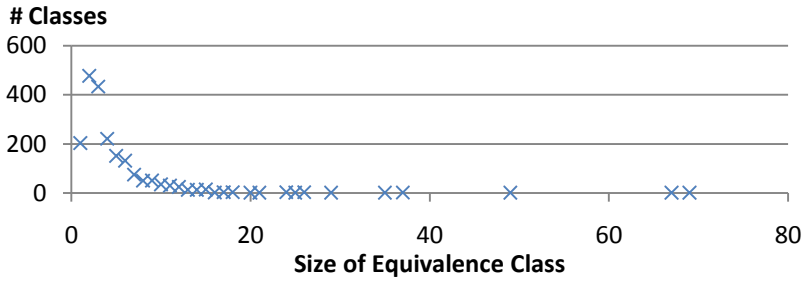
3.5.1 First Order Control Flow

One of the properties we consider hard to modify is the order in which the code gets executed. Therefore, we collect information about the possible flow of control through the program. Every control transfer between instructions in the program is recorded. As such, we collect first order control flow.

In practice, this results in a control flow graph that may contain unrealizable paths: paths containing more than one control transfer may be represented by the graph, but never executed in practice. This is illustrated in Figure 3.7. Assume that the execution trace is A-C-D-B-C-E, then the first order control flow information is represented by the graph. However, this graph also represents the (sub)path A-C-E, which has not been observed and may even be unrealizable.



(a) False positive rates



(b) Equivalence classes at ins level

Figure 3.6: Evaluation of the classifier based on first execution time

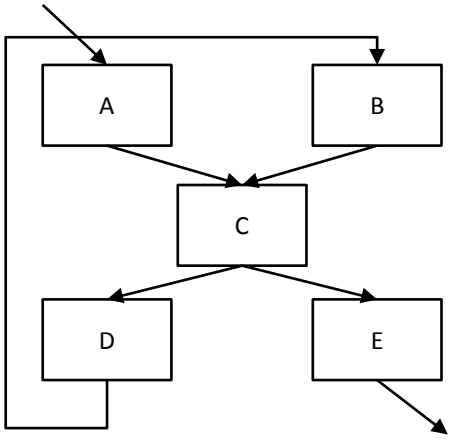


Figure 3.7: Unrealizable paths in first order control flow

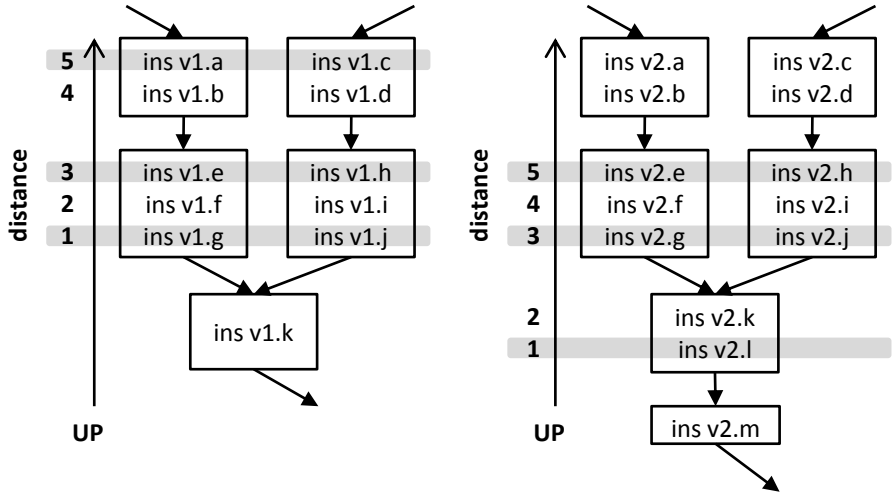
The classifier based upon this type of information has two parameters: distance and direction. The direction indicates whether to go up, down or in both directions in the control flow graph. The distance indicates how many instructions to ascend and/or descend. When there are multiple outgoing and/or incoming edges, all of them are traversed. This is illustrated in Figure 3.8. For the purpose of illustration, instructions which are supposed to match have been given the same coordinates in version 1 and version 2. The difference between the two versions could have been obtained by inserting two garbage instructions in version 2.

The obtained sets of instructions are shown in Figure 3.9. The arrows indicate instructions that have previously been matched. Six out of the ten instructions of the first set have a match in the second set, six out of eight instructions of the second set have a match in the first set. The score is then computed as follows: $\frac{\frac{6}{10}+\frac{6}{8}}{2} = \frac{27}{40}$.

Evaluation

As this classifier requires previously matched instructions, its strength will depend on how well matches have been found in earlier iterations of the matching system. To eliminate this dependency on earlier iterations, we have evaluated the classifier assuming that all pairs of code fragments, except for the pair under consideration, have been classified correctly. The settings of the classifier for the purpose of this evaluation are direction=both and distance=3.

As can be seen in Figure 3.10, this classifier has significantly more discriminative strength than the classifier based on instruction syntax, both at the instruction and at the basic block level of granularity. At threshold 1, the false



(a) version 1 (b) version 2

Figure 3.8: Control flow graph traversal for instructions ins_v1.k and ins_v2.m with direction=up and distance=5

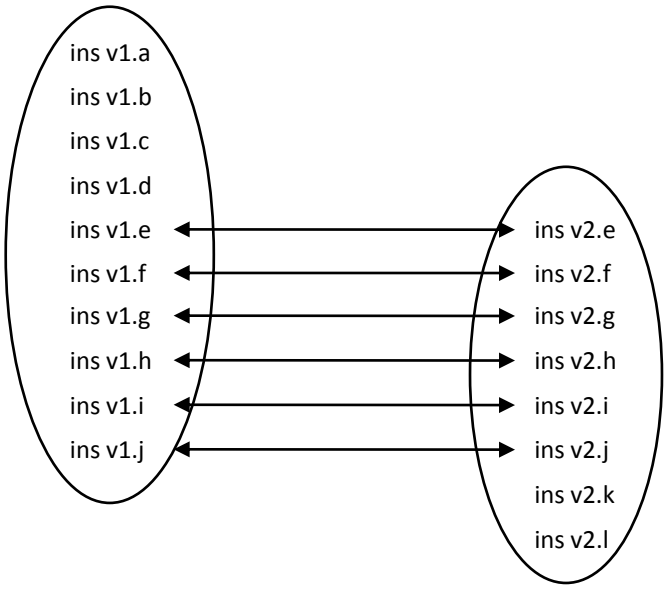


Figure 3.9: Comparing instruction sets

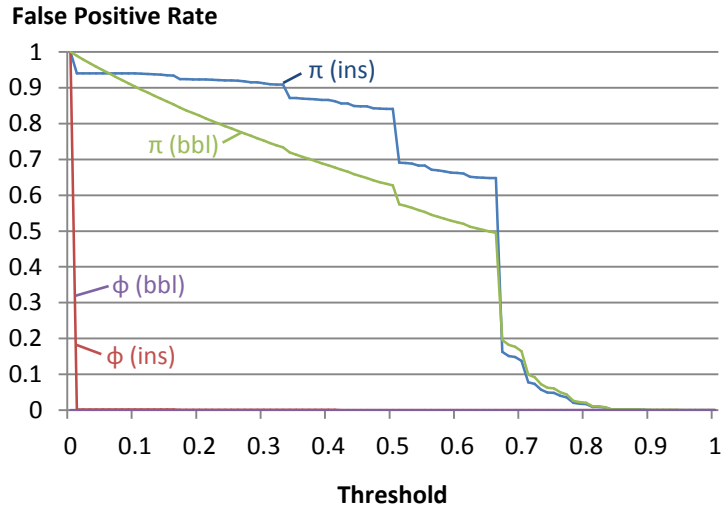


Figure 3.10: Evaluation of the classifier based on control flow with direction=both and distance=3

positive rate is almost 0. At the instruction level, all but four instructions are in an equivalence class of size 1. At the basic block level, all but 43 basic blocks are in an equivalence class of size 1.

As such, this classifier will be very useful in improving an existing relatively good mapping, by filtering matches that are not related when looking at their context in the control flow graph, or by extending correctly matched regions along the flow of control.

3.5.2 First Order Data Flow

Similar to control flow, we keep track of first order data flow. This is typically represented by a data dependency graph. In a data dependency graph, edges connect instructions to the instructions that last defined the values used. Similarly to control flow, we consider data flow to be particularly hard to modify for programs in general.

The classifier operates similarly to the classifier operating on first order control flow.

Evaluation

Many of the comments made in the evaluation of the classifier based on first order control flow are valid for the classifier based on first order data flow

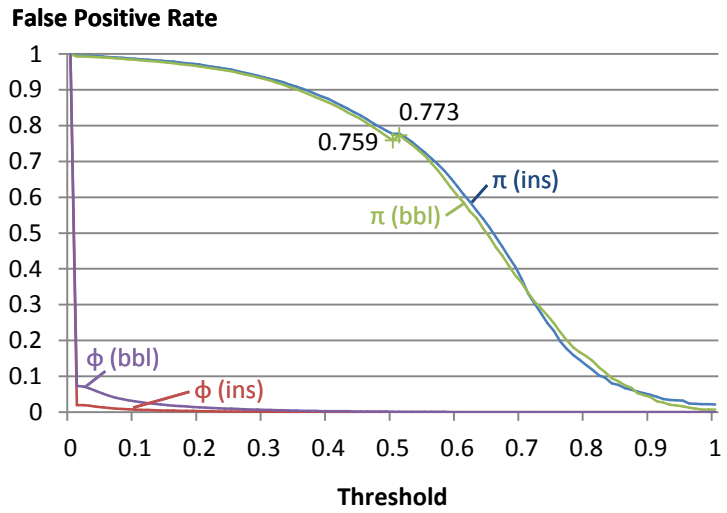


Figure 3.11: Evaluation of the classifier based on data flow with direction=both and distance=3

as well. Furthermore, we have used the same settings, direction=both and distance=3.

The results (Figure 3.11) illustrate that π (as opposed to ϕ) is not necessarily decreasing as the threshold increases. The numerator for both π and ϕ is the absolute number of false positives, which is non-increasing. As the denominator is constant for ϕ, ϕ is non-increasing as well. However, for π, the denominator is the size of the estimated mapping, which is non-increasing. When passing threshold 0.5, we lose 46,205,330 false positives, but we also lose 1925 correctly identified matches.

Similarly, ν is non-decreasing as the threshold increases.

Again, at threshold 1, the false positive rate is almost 0. At the instruction level, we observe only 71 instructions in equivalence classes with sizes greater than one. At the basic block level, all but 9 basic blocks are in an equivalence class of size 1.

3.6 Building a Matching System from Fuzzy Classifiers

The fuzzy classifiers discussed in the previous sections each indicate their confidence as to whether two code fragments are related. When evaluating the success of matching two identical versions, it already becomes apparent that the false positive rate is unacceptably high, even at threshold 1, with the exception

of the proximity-based classifiers and the classifier based on first execution time.

The evaluation of the proximity-based classifiers is good because we have assumed a perfect mapping of all of the code fragments, excluding those under consideration. Clearly, this assumption is not viable in a real setting. Furthermore, the classifier based on first execution time is very vulnerable at lower thresholds, as can be observed from Figure 3.6. This indicates that it will not be as successful once we are dealing with truly diversified versions and are forced to pick a lower threshold.

We have taken three approaches to mitigate the problem of unacceptably high false positive rates: combination and iteration of fuzzy classifiers and limiting the number of matches per code fragment.

3.6.1 Combining Fuzzy Classifiers

Let's assume that the event that two unrelated code fragments are perceived similar by one classifier c is independent of the event that they are perceived similar by another classifier d, i.e.,

$$\begin{aligned}&\forall (A,B) \in \mathcal{P}(C)^2, \forall (x,y) \in [0,1]^2, \forall a \in A, \forall b \in B : (a,b) \notin \mu_r \\ &\Rightarrow P[c(a,b) \geq x \wedge d(a,b) \geq y] = P[c(a,b) > x]P[d(a,b) > y] \ .\end{aligned}$$

In that case, the probability of two unrelated code fragments being considered similar by two classifiers at the same time is lower (except when both probabilities are one) than the probability that they are considered similar by only one classifier.

Clearly, this is an oversimplification. Yet, in practice, we can observe that the correlation is still small enough to reduce the number of false positives through the combination of fuzzy classifiers: unrelated code fragments may often seem related with respect to a single classifier, but they rarely seem related when looking at multiple classifiers at the same time.

Note that if one of the selected classifiers has no opinion on a pair of code fragments, the combined classifier will have no opinion either.

Evaluation

We have illustrated this phenomenon in Figure 3.12. Here, we have plotted π at the granularity of basic blocks. When looking at instruction syntax alone, π, even at threshold 1, is still around 0.8 (see Figure 3.2). Similarly, it was at 0.86 for the classifier based on data (see Figure 3.3). When they are combined, we can observe far better results, with π around 0.16 when both classifiers are set at threshold 1.

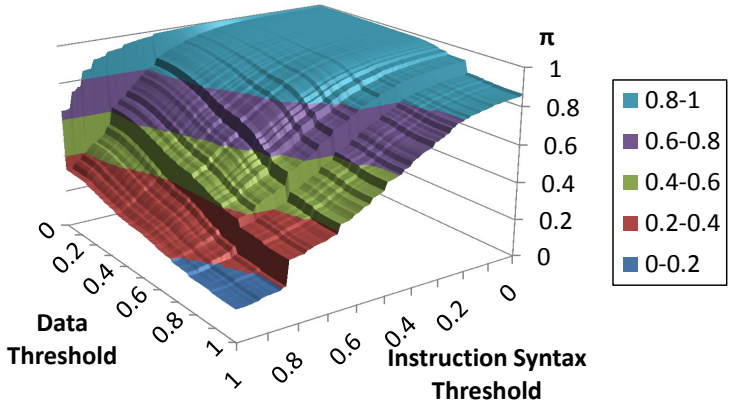


Figure 3.12: Evaluation of the diversity system composed of the classifiers based on instruction syntax and data

We can observe the same graph as in Figure 3.3 for π (bbl) when the classifier based on instruction syntax is set to threshold 0. The same does not hold for the classifier based on instruction syntax, because by combining it with the classifier based on data, the combined classifier no longer has an opinion on a significant number of pairs, even at threshold 0.

3.6.2 Limiting the Number of Matches

From the number of code fragments in equivalence classes of size n, we can conclude that a large fraction of the false positives is the result of code fragments in large equivalence classes: an equivalence class of size n leads to $n \times n - n$ false positives. Assuming that there will be a limited number of related code fragments between versions, a first protective measure is to limit the number of matches per code fragment.

Therefore, we have added the option to match only the most related code fragment during an iteration. As the basis of comparison may be a tuple of values when different classifiers are combined, we do not have a total order relation, but a partial order relation. Currently, the best (worst) candidate is the first encountered candidate on the Pareto front.

Another parameter determines the maximum number of matches per code fragment.

3.6.3 Iterating Fuzzy Classifiers

The quality of the proximity-based classifiers depends heavily upon the quality of the existing mapping. In the earlier evaluation, we have assumed that a perfect mapping was available for all pairs of code fragments with the exception of the pair under consideration. Clearly, this assumption is not viable in a real-life scenario. Therefore, we will need to obtain an initial mapping during an earlier iteration, before we can use these classifiers in subsequent iterations.

The final mapping of a matching system will therefore be obtained in a number of iterations. Each iteration either introduces new matches between code fragments or removes existing matches between code fragments.

During the first iteration, while producing the initial estimate, no existing matches are available. As a result, classifiers that require a preexisting preliminary mapping (the proximity-based classifiers) are not available during the initial estimate. In subsequent iterations, any combination of fuzzy classifiers is available to either add pairs of code fragments to the estimated mapping (an extend iteration) or to remove pairs of code fragments (a filter iteration).

During an extend iteration, only pairs of code fragments (a_i, b_j) for which every selected classifier c returns a value higher than or equal to the associated threshold can be added. Similarly, during a filter iteration, only pairs of code fragments (a_i, b_j) for which every selected classifier returns a value lower than or equal to the threshold can be removed.

The "no opinion" value behaves as the value 0 during an extend phase and as the value 1 during a filter phase. As a result, pairs for which the (combined) classifier does not have an opinion will not be added or removed from the reference mapping for reasonable confidence thresholds (in the interval $]0, 1]$ during an extend phase and in the interval $[0, 1[$ during a filter phase).

The Default Matching System

In order to have a uniform basis for comparison, we put forward a matching system defined by the iterations shown in Table 3.1.

The other settings are fixed over the different iterations: we look at basic block granularity and add only the best matches. For simplicity, we will refer to this matching system as the default matching system.

Evaluation

When we apply this matching system on two identical copies of the mcf benchmark, we obtain the false positive and false negative rates as shown in Figure 3.13. After the last iteration, we have a false negative rate of about 0.8% and a false positive rate of about 2.5%.

	General Settings		Classifier Settings			
Iteration	Phase	#Matches	Classifier	Threshold	Direction	Δ
1	Init	1	Syscalls	0.3		
2	Extend	1	Syntax	0.5		
			Data	0.5		
			Order	0.5		
			Freq.	0.5		
3	Extend	2	Syntax	0.1		
			CF	0.1	BOTH	3
4	Filter		CF	0.1	UP	3
5	Filter		CF	0.1	DOWN	3
6	Filter		DF	0.1	UP	3
7	Filter		DF	0.1	DOWN	3
8	Extend	2	Data	0.7		
			DF	0.3	BOTH	3
9	Extend	2	Syntax	0.7		
			CF	0.3	BOTH	3

Table 3.1: Settings of the default matching system

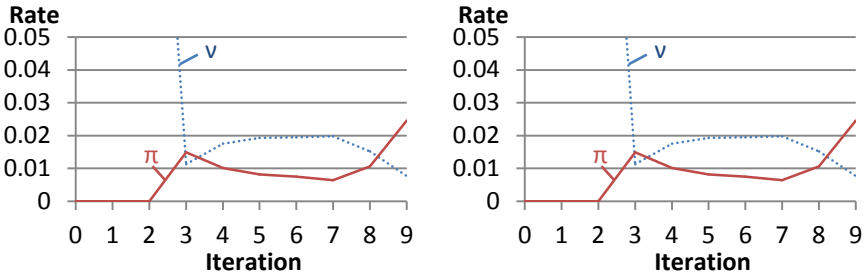
Note that we could easily get a perfect match in one iteration using the classifier based on first execution time at threshold 1. However, we have included the evaluation with the default diversity system as a baseline for comparison to the diversifying transformations in the next chapter.

3.7 Related Work

Matching two versions of some piece of information has many applications, ranging from text file comparison to DNA matching. In this section, we briefly discuss related work relevant to the matching of program versions.

3.7.1 Text-based Matching Approaches

Text-based matching algorithms are about finding the minimum script of symbol deletions and insertions that transform one sequence into another. This type of matching is used in, e.g., spelling correction systems and file comparison tools [Miller 85, Wagner 74]. However, due to the limitations on the types of changes (insertions and deletions), they cannot fully capture the degrees of freedom in the language of programs. While this typically is not a problem



(a) Full view (b) Zoomed

Figure 3.13: Evolution of the false positive and negative rate for the trivial diversity system

in the application domain of text-based matching algorithms, it does become problematic when, e.g., statements have been deliberately reordered.

3.7.2 Graph-based Matching Approaches

More recently, graph-based binary matching algorithms [Dullien 05, Sabin 04] have been proposed, which compare high-level structures like control flow graphs, as opposed to source code, assembly or code bytes. Often this offers a more illuminating view of relevant differences between two program versions. This approach can also foil diversification methods that do not alter the control flow graph much, e.g., reordering transformations.

3.7.3 Trace-based Matching Approaches

Trace-based matching approaches collect information about the execution of the program, such as control flow, values produced, addresses referenced and data dependencies exercised [Zhang 05]. They have been evaluated by comparing unoptimized and optimized versions of a program. Recently, they have been used to compare original and obfuscated versions as well [Nagarajan 07].

3.7.4 Matching Tools

Different tools have been developed with different goals in mind built upon one or more of the previously discussed matching techniques. BMAT [Wang 00] is a binary matching tool that has been developed with the primary goal of reusing profile information in subsequent builds.

Other tools, such as BinDiff[1] and the eEye Binary Diffing Suite[2] are targeted more specifically at pinpointing differences between different versions. Both tools are extensions to the popular IDA Pro Disassembler and Debugger,[3] and are primarily targeted at analyzing patches.

[1] http://www.sabre-security.com/products/bindiff.html
[2] http://research.eeye.com/html/tools/RT20060801-1.html
[3] http://www.datarescue.com/idabase/

4

Diversity System – Proteus

The central part of every application of software diversity is a system to generate semantically equivalent, but syntactically different code: a diversity system.

We will define a diversity system as follows. If C denotes the set of all syntactically correct code fragments in whatever language the code is specified, a diversity system D takes as input a fragment of code $c \in C$ and a set of nonces (numbers used once) $\{1, \ldots, k\}$ and produces a set of code fragments $\{D(c,1), \ldots, D(c,k)\}$ so that $\forall i \in \{1, \ldots, k\} : D(c,i)$ has the same functionality as c, yet $\forall (i,j) \in \{1, \ldots, k\}^2, i \neq j \Rightarrow D(c,i)$ is syntactically different from $D(c,j)$.

Similar to Kerckhoffs' principle for cryptography (a cryptosystem should be secure even if everything about the system, except the key, is public knowledge), we must assume that everything about the system, excluding the used nonces (numbers used once) is public knowledge. We should also assume that an attacker will have access to one or more of the diversified versions. This is illustrated in Figure 4.1.

4.1 Combining Diversifying Transformations

In order to fool the type of matching system discussed in the previous chapter, we need to avoid any type of information that can identify related code fragments accurately. In practice, it proves to be difficult to design a single

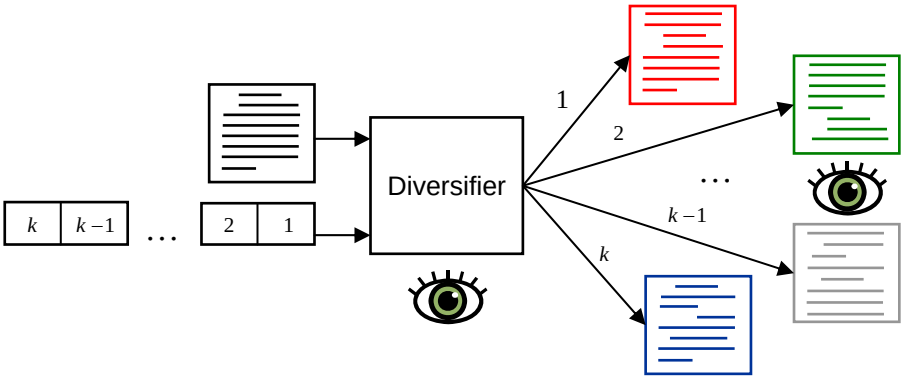


Figure 4.1: Extended schematic of a diversity system

monolithic transformation which removes discriminative invariants for the different types of information at the same time.

However, many transformations each affect one type (or a limited number of types) of information. By combining many of these "smaller" transformations affecting different types of information, we can increase the range and complexity of randomization. While such primitives may be insecure when used alone, iterated application can create complexity, including emergent properties due to interaction among various transformations. This is similar to behavior found in complex systems such as cellular automata, and also helps to create confusion and diffusion, as in iterated application of rounds in block ciphers and hash functions.

Increasing the Range

The goal of combining these transformations is to enlarge the set of semantically equivalent fragments of code that can be generated by the resulting diversity system D from a fragment of code $c \in C$. As usual this is referred to as the "range" property, $\mathrm{ran}(D, c)$.

The cardinality of the range of a diversity system can easily become very large. Consider the diversity system which chooses for every instruction in the original code whether or not to precede it by a nop-instruction. If that fragment of code consists of n instructions, then the cardinality of the range is 2^n. This diversity system has a large range, yet has little merit for most applications of diversity. Therefore, the cardinality of the range alone is not a good indication of the quality of a diversity system.

On the other hand, a diversifier E of which the range is a superset of the range of another diversifier D ($\forall c \in C : \mathrm{ran}(D, c) \subseteq \mathrm{ran}(E, c)$) will typically

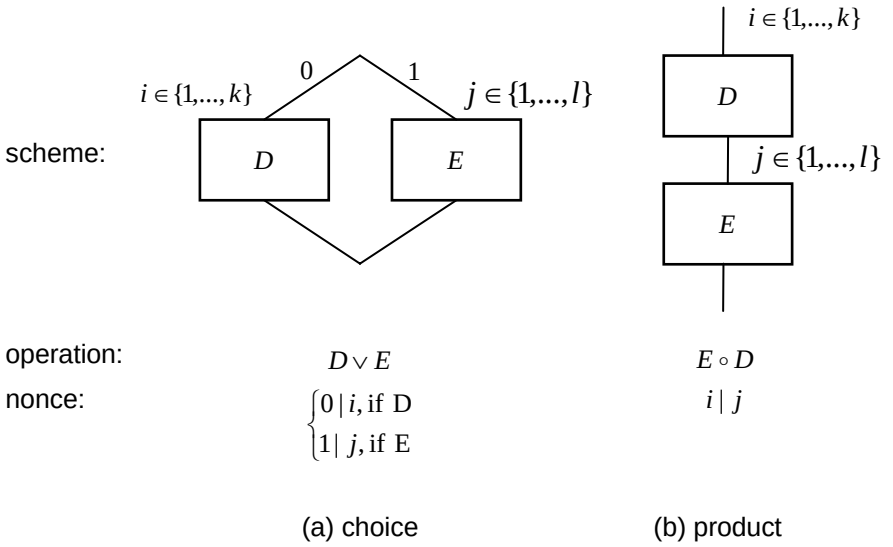


Figure 4.2: Two combining operations for diversifying transformations

be preferred, as this indicates that more diversity can be achieved. We will abbreviate this relation as follows: $D \leq E$

Choice Operation

Fortunately, given two diversity systems D and E, it is easy to create a third diversifier F for which $\mathrm{ran}(D) \subseteq \mathrm{ran}(F)$ and $\mathrm{ran}(E) \subseteq \mathrm{ran}(F)$ through the choice operation (Figure 4.2(a)): $F = D \vee E$. This corresponds to making a preliminary choice as to whether system D or E is to be used. When this is done, D or E is used as originally defined. Note that $D \vee E = E \vee D$.

Product Operation

A second combining operation (Figure 4.2(b)): $F = E \circ D$, corresponds to diversifying the program with the first diversifier D and diversifying the resulting program with the second diversifier E, the nonces for D and E being chosen independently. This total operation is a diversifier whose transformations consist of all the products (in the usual sense of products of transformations) of transformations in E with transformations in D.

Logging the Applied Transformations

Keeping track of the applied transformations is important as some applications may need to be able to recreate the diversified copies. Consider for example randomizing programs before distribution. When updates are needed afterwards, the software provider may need to tailor them to a specific copy. Maintaining a database of nonces requires less storage than keeping a copy of every distributed version.

This choice operation is recorded in the resulting nonce as follows: if $\{1, \ldots, k\}$ (respectively $\{1, \ldots, l\}$) is the range of nonces accepted by D (respectively E), then for $i \in \{1, \ldots, k\}, j \in \{1, \ldots, l\}$ the nonce becomes $0|i$ or $1|j$, where '|' is the concatenation operator. The nonce of the product operation then becomes $i|j$.

4.2 Determining the Reference Mapping

In order to evaluate the quality of an estimated mapping μ_e, we need a reference mapping μ_r. To this end, each of the transformations has been designed to maintain a mapping to the addresses of the original instructions. As a result, every instruction has a set of original addresses. Instructions from version 1 which have an original address in common with instructions from version 2 are considered to be related.

This is illustrated in Figure 4.3. Here, two instructions of the original program O_i and O_j have been merged into a single instruction in version 1: a_i. Therefore, this instruction in version one will have a set of two original addresses, i.e., $a_i : \{O_i, O_j\}$. In version 2, the two original instructions are still separate. Therefore, they will each have a single original address, i.e., $b_i : \{O_j\}, b_j : \{O_i\}$. These sets of original addresses are then used to determine the correct mapping between the two versions: $\mu_r = \{(a_i, b_i), (a_i, b_j)\}$. This mapping will determine the number of false positives and false negatives in the estimated mapping.

4.3 Syntactically Different Versions

An important property of a diversity system is that the resulting programs are in fact diverse. Therefore, we want different nonces to lead to different programs.

Nonce-injective

A diversity system is said to be *nonce-injective* if:

$$\forall c \in C, \forall i, j \in \{1, \ldots, k\} : i \neq j \Rightarrow D(c, i) \neq D(c, j) \ .$$

Typically, this property will hold for basic transformations (not a composition of other transformations). It may however become an issue when many transformations are combined using the combination operations described earlier, as the product of two nonce-injective transformations is not necessarily nonce-injective.

Injective

A transformation is *injective* in the traditional sense if:

$$\forall (c_1, c_2) \in C^2, \forall (i, j) \in \{1, \ldots, k\}^2 : \\ c_1 \neq c_2 \vee i \neq j \Rightarrow D(c_1, i) \neq D(c_2, j) \ .$$

Clearly, the composition of two injective transformations is injective. Furthermore, if E is an injective transformation and D is a nonce-injective transformation, then $E \circ D$ is nonce-injective. Note that if c_1 and c_2 have different semantics, c_1 cannot be syntactically equal to c_2. This definition is therefore only useful when c_1 and c_2 are two semantically equivalent, but syntactically different versions of a fragment of code (e.g., after applying a diversity system).

Disjoint

We say that two diversity systems D and E are *disjoint* if and only if

$$\forall c \in C, \forall i \in \{1, \ldots, k\}, \neg \exists j \in \{1, \ldots, l\} : D(c, i) = E(c, j) \ .$$

The choice of two disjoint injective transformations ($D \vee E$) is injective.

4.4 Diversity Systems in Practice

The injective property and related properties discussed earlier prove to be a useful guideline in the selection of transformations to add to the mix. For example, it is not useful to add a transformation D to a diversity system E if the range is not increased as a result ($D \leq E$).

Clearly, injective transformations disjoint with the already present diversity system are preferred. However, in practice, this requirement is not so stringent. Because of the large range, the probability of actually obtaining two identical code fragments after a number of transformations is small. If required, a hash can be computed of every generated code fragment and newly generated code fragments can simply be discarded if their hash matches one of the earlier ones.

A practical diversity system may be composed of a number of transformations: $(D_1 \vee D_2 \vee \ldots \vee D_n) \circ (D_1 \vee D_2 \vee \ldots \vee D_n) \circ \ldots \circ (D_1 \vee D_2 \vee \ldots \vee D_n)$.

Conditions on Transformations

The probabilities which determine which transformation to choose can be made assignable, and change as the result of earlier transformations. It may, for example, be useless to apply the same transformation twice, which can be recorded by setting its probability to zero. Furthermore, some transformations may no longer be possible once other transformations have been applied. Likewise, some transformations may require other transformations to have taken place. These dependencies can be represented by postconditions and preconditions. For a more elaborate discussion on the selection of transformations with dependencies, we refer to closely related work on selecting transformations in the domain of obfuscation by Heffner and Collberg [Heffner 04].

Selecting Nonces

In practice, it proves to be complicated to determine the range of nonces accepted by a composed diversity system. The application of one transformation will lead to more or less possibilities for the next transformation in a way that is hard to predict without actually applying the transformation. As the range quickly becomes unmanageable, generating all possibilities to determine the range in advance is also not practically viable. Therefore, we cannot predetermine a uniform range of nonces from which to choose in advance. Rather, every transformation will return its range once it is selected as the next transformation (and all previous transformations have been applied), after which an element from its range is selected. The nonces are thus built dynamically during the randomization as shown in Figure 4.2 and can have variable lengths.

4.5 Experimental Setup

In our experiments, the nonce selection is guided by a pseudorandom number generator. Most experiments are based on two versions generated from two different seeds. Section 4.7 contains a more extensive study with 50 different seeds to show that the results are relatively independent of the chosen seeds.

Benchmarks

In this chapter, we will continue to work with the mcf benchmark of the SPEC CPU2006 benchmark suite. We will study the impact of the transformations on the other benchmarks of the suite in Section 4.7 to indicate the dependence of the results on the benchmark under consideration.

The other C benchmarks in the evaluation suite will also be used to indicate the ranges of the transformations. A brief description of the functionality of these benchmarks is provided in Table 4.1.

We have reported the number of functions, basic blocks and instructions in these benchmarks, and the number of those that were executed at least once during the test runs in Table 4.2. These figures can be used to relate the number of candidates for the described transformations to the number of code fragments of a given type. From this table, we can learn that gcc, perlbench and gobmk are the largest benchmarks in terms of function, basic block and instruction count. Furthermore, gobmk seems to execute a lot of small functions.

Benchmark	Description
400.perlbench	PERL programming language
401.bzip2	Compression
403.gcc	C compiler
429.mcf	Combinatorial optimization
433.milc	Physics: quantum chromodynamics
445.gobmk	Artificial intelligence: go
456.hmmer	Search gene sequence
458.sjeng	Artificial intelligence: chess
462.libquantum	Physics: quantum computing
464.h264ref	Video compression
470.lbm	Fluid dynamics
482.sphinx	Speech recognition

Table 4.1: Description of the C programs in the SPEC CPU2006 benchmark suite

Cost of the Transformations

When reporting timing results, they represent the average over three different, non-consecutive runs on an otherwise idle system. The impact on the code size is measured taking into account only the size of the code sections, as all of our transformations are code transformations.

4.6 Diversifying and Anti-tampering Transformations

The main goal of diversifying transformations is to introduce a high level of diversity between the different versions. This will be measured by how well they succeed in thwarting the matching techniques discussed in the previous chapter.

	static			executed		
benchmark	# fun	# bbl	# ins	# fun	# bbl	# ins
400.perlbench	2428	84375	304978	829	13578	52484
401.bzip2	855	22475	91097	145	2589	13609
403.gcc	5334	214343	766059	2339	67566	254761
429.mcf	814	20440	80380	178	1950	8494
433.milc	979	24433	99557	249	3564	17994
445.gobmk	3329	55301	238493	1273	5444	22351
456.hmmer	1077	27988	112283	257	3286	14855
458.sjeng	909	25745	100242	218	3822	15638
462.libquantum	854	21407	85153	158	1735	8721
464.h264ref	1310	40624	191885	456	7733	40712
470.lbm	821	20689	82355	160	1463	7866
482.sphinx	1047	27044	109870	373	6050	28118

Table 4.2: Static and dynamic function, basic block and instruction count

In general, there are two ways to thwart a matching system. Firstly, we can make it harder to obtain an estimated mapping. This can be done by undermining assumptions made by the attacker or by effecting a slowdown on the used algorithms.

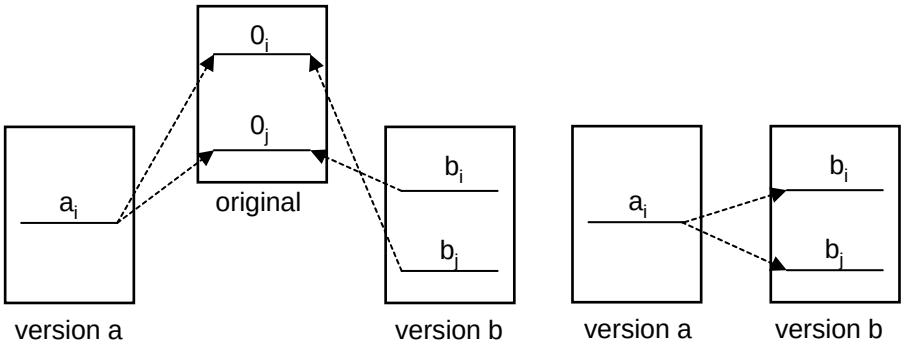
Secondly, we can make the estimated mapping less accurate. This can be done either by making unrelated code fragments seemingly related (increasing the false positive rate) or by making related code fragments seemingly unrelated (increasing the false negative rate).

Besides introducing diversity between versions, we need to take into account the tamper resistance of the individual copies as well. Therefore, where appropriate, we will highlight choices that can increase the tamper resistance of individual copies.

Parameterizing Transformations

In this section, we will study the applicability of a number of existing transformations from different domains within the context of software diversity. These transformations are taken from the domain of code optimization (factorization and unrolling), code obfuscation and code generation.

In particular, we will discuss how they can be parameterized to lead to syntactically different programs. In general, we distinguish between two types of transformations. The first type are transformations for which we can only choose whether or not to apply them. In these cases, we can set the probability p with which we will apply the transformation. Depending on the outcome of



(a) Old addresses of version a and b (b) Correct mapping

Figure 4.3: Determining the reference mapping and the impact of folding

the pseudorandom number generator, the transformation will then be applied or not.

The second type of transformations allow us to choose between different alternatives. For this type of transformation we will randomly select one of the alternatives based on the pseudorandom number generator.

4.6.1 Folding

Folding transformations reduce the number of (almost) identical code fragments within the program by reusing one of them in the different contexts in which the (almost) identical code fragments originally appeared.

Folding transformations can help at foiling matching algorithms because they invalidate the assumption that a code fragment from one version has at most one corresponding code fragment in another version. This is illustrated in Figure 4.3. Part (a) illustrates the original program, which contains a pair of instructions that can be folded. In version a, we choose to fold them, while we choose not to fold them in version b. As a result, the folded instruction in version a corresponds to two instructions in version b, as illustrated in part (b).

The assumption that the number of matches per code fragment is limited can significantly reduce the false positive rate generated by a couple of very large equivalence classes, as discussed in the previous chapter. If the number of code fragments in version a is given by n_a, and for version b by n_b, then there can be at most $\min(n_a, n_b)$ false positives if the number of matches per code fragment is limited to 1. When folding transformations have been applied, the number of matches per code fragment is no longer limited to 1. If we would be unable to impose any limitation, the number of false positives can be of the order of $n_a \times n_b$.

Folding is furthermore an excellent form of tamper resistance. An attacker typically wants to obtain a *small* semantic change of the software through a *small* syntactical change. As discussed in the case studies of Section 1.1.1, the small semantic change is aimed at turning the behavior intended by the software provider into the behavior desired by the attacker. A small syntactical change allows the attacker to tamper with the program even with a very narrow view on a limited aspect of the program.

Folding operations increase code reuse from different contexts and therefore a small change may affect many conceptually hardly related aspects of the software. As a result, after folding operations, a *small* syntactical change may result in a *large* semantic change of the program. An attacker generally does not want a large change in the behavior of the software: if that were the case, he would not be interested in the software in the first place and may be better off rewriting the software from scratch.

Unfortunately, the available range of folding transformations is inherently limited by the redundancy of the original program. Diversity can be obtained by selecting different subsets of the folding candidates for different versions.

Function Factoring

We can choose for each list of identical functions whether or not to factor them. Identical functions can originate from, a.o., copy-paste behavior, C++ templates and small wrapper functions.

Epilogue Factoring

We can choose for each list of identical function epilogues whether or not to factor them. Function epilogues are the return basic blocks of a function. These can be factored easily because we do not need to make special precautions to redirect control flow depending on what context they are called from: the `ret` instruction simply fetches the address from the stack.

Basic Block Factoring

We can choose for each list of basic blocks with identical bodies whether or not to factor them. The body of a basic block is obtained by stripping the basic block from its control transfer instruction. These blocks can be factored by using the call/return mechanism to ensure correct transfer at the end of the basic block back to the context from which it is executed.

Evaluation

We have indicated the number of opportunities for the folding transformations in Table 4.3. For perlbench, e.g., we have 182 pairs of identical functions (f_i, f_j) and we can choose for each pair whether or not to redirect all calls to f_j to f_i and to remove f_j. This choice can be made independently for each pair, resulting in a range of 2^{182} different programs.

benchmark	function	epilogue	basic block
400.perlbench	182	208	3383
401.bzip2	46	71	1029
403.gcc	1230	311	9321
429.mcf	46	69	954
433.milc	121	89	1135
445.gobmk	15188	239	2338
456.hmmer	50	98	1267
458.sjeng	47	81	1327
462.libquantum	49	68	1010
464.h264ref	54	114	1977
470.lbm	51	67	989
482.sphinx	52	97	1181

Table 4.3: Number of candidates for folding per benchmark

In Figure 4.4, we have plotted the cost of the transformations in terms of code size and execution time. On the x-axis, we have assigned different values to p, the probability with which a folding transformation is applied. We have chosen to use the same p for the three transformations at the same time.

Function and epilogue factoring require no additional instructions to be executed. The additional instructions required for basic block factoring have little impact on the execution time. A small improvement in code size can be observed when the transformations are applied more often. This is in the line of the expectations of factoring operations. However, as the goal of the folding operations is not compaction, but diversity, we also allow folding even if there is no net gain. Hence, the code size decrease would be even bigger if the folding operations were optimized for that criterion.

Given a pair of candidates for folding, the probability that they are only selected in one version (and not in the other one) is given by $2 \times p(1-p)$. As we want to maximize this probability, we have set $p = 0.5$.

The impact of the folding transformations on our default matching system is shown in Figure 4.5. After the last iteration, we have a false negative rate of about 5% and a false positive rate of almost 10%. This indicates that the

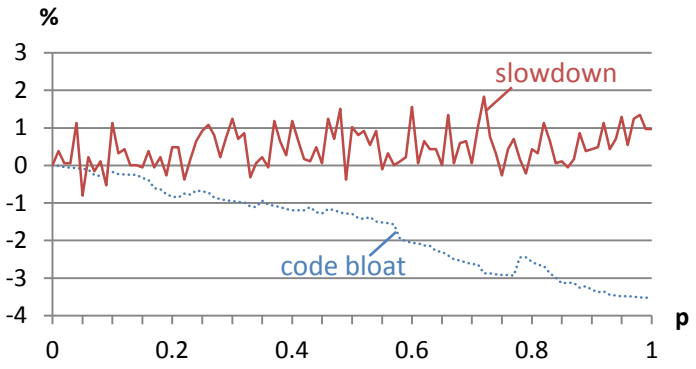


Figure 4.4: Code bloat and slowdown for the folding transformations

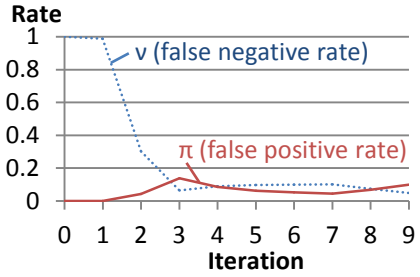


Figure 4.5: Evolution of the false positive and negative rate for the combination of the three folding transformations (p=0.5)

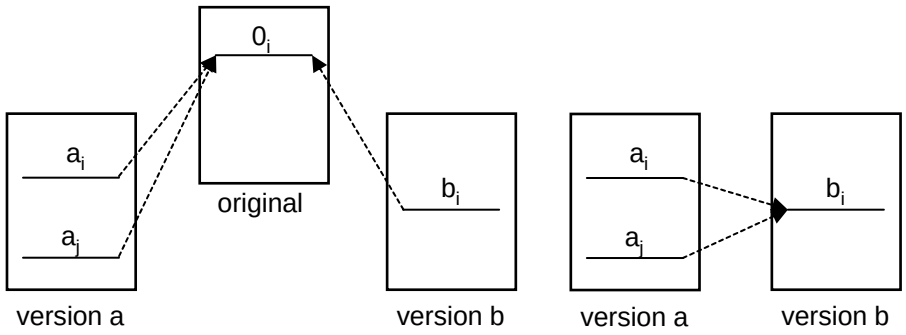
matching system has considerably more difficulty to match all code fragments correctly, despite the very low cost of the transformations in terms of execution time and even a gain in terms of code size.

4.6.2 Unfolding

Unfolding transformations increase the number of identical code sequences within the program by using a different copy depending on the context where the original sequence appeared.

Unfolding operations can typically be applied repeatedly. For example, a loop can be unrolled infinitely and recursive function calls can be inlined infinitely. Diversity can be obtained by selecting different (code fragment, context) pairs to unfold.

Similarly to folding transformations, unfolding transformations invalidate the assumption that an instruction from one version has at most one corresponding instruction in another version. This is illustrated in Figure 4.6. In



(a) Old addresses of version a and b (b) Correct mapping

Figure 4.6: Impact of unfolding on the reference mapping

version 1, an instruction was duplicated during an unfolding transformation, while this transformation was not applied in version 2. As a result, two instructions from version 1 correspond to the same instruction in version 2.

Unfolding is likewise an excellent form of tamper resistance. As mentioned earlier, an attacker wants to obtain a *small* semantic change of the software through a *small* syntactical change. As a result of the unfolding transformations, a *larger* syntactical change may be needed to effect a *small* semantic change of the program.

Function Inlining

If a function is called from multiple sites, we can choose for each call-site whether or not to inline the function at the call-site.

Basic Block Unfolding

If a basic block has multiple incoming edges, we can choose for each incoming edge whether or not to duplicate the block and to redirect the incoming edge to the newly created block. This is illustrated in Figure 4.7.

Two-way Opaque Predicating

We can choose for each basic block whether or not to duplicate it and to choose between one of the two alternatives based upon a two-way opaque predicate. A two-way opaque predicate is a predicate that can evaluate to both true and false during the execution. For each selected basic block, a two-way opaque predicate can be chosen from a library of two-way opaque predicates [Collberg 98b]. In this particular transformation, we thus artificially

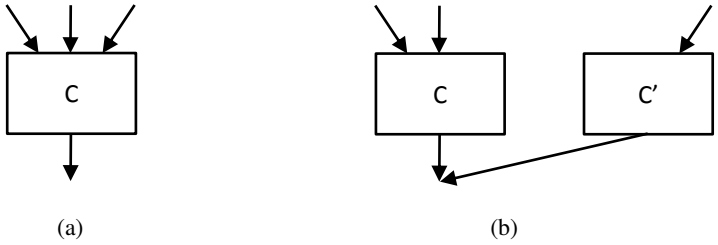


Figure 4.7: Inlining a basic block at the incoming edge

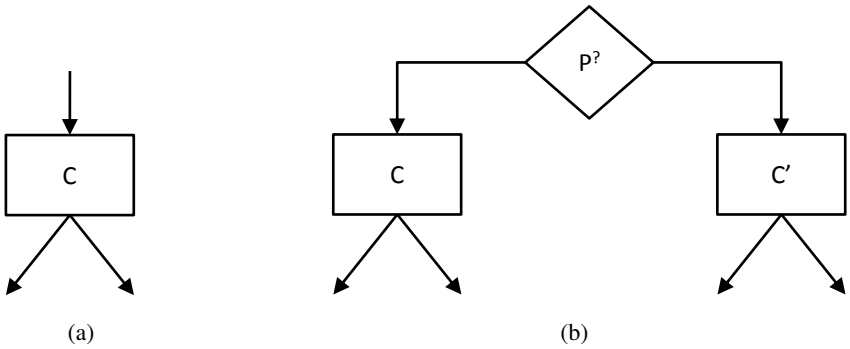


Figure 4.8: Predicating a basic block by a two-way opaque predicate

create an additional context from which to transfer control to (a copy of) the basic block. This is illustrated in Figure 4.8.

Evaluation

The range of the different unfolding transformations is shown in Table 4.4, when applied during a single iteration to the original program. As already mentioned, we could reapply these transformations: unfolding a loop can be done indefinitely, leading to an infinite range. Furthermore, the range of two-way opaque predicates only takes into account the candidates for predicating, not the choice between different predicates.

As could be expected, these ranges are significantly larger than those of the folding transformations. The cost of these transformations with respect to code size is also higher. However, the cost of unfolding results in no adverse effect for the matching system when it is done on code that is never executed (frozen code). Therefore we will use profiling information (from the training input sets) to avoid this type of code.

	No restrictions			With restrictions		
benchmark	fun	bbl	2way	fun	bbl	2way
400.perlbench	8052	54071	83759	1063	23980	16938
401.bzip2	1377	12646	22269	43	2441	2195
403.gcc	19995	150632	212431	5040	85106	54097
429.mcf	1299	11425	20240	64	2309	1792
433.milc	2139	13986	24212	305	3729	3278
445.gobmk	6527	34016	54875	2193	17660	18004
456.hmmer	2557	16366	27754	297	4279	3238
458.sjeng	1858	14813	25508	230	3951	3294
462.libquantum	1397	12050	21194	59	2016	1648
464.h264ref	2666	22692	40362	442	7286	7461
470.lbm	1315	11571	20418	45	1958	1368
482.sphinx	2730	15688	26794	759	5488	5843

Table 4.4: Number of candidates for unfolding per benchmark. Without restrictions and with restrictions to (i) non-frozen code for function and basic block inlining, (ii) lukewarm code for two-way opaque predicating

With the exception of two-way opaque predicating, which introduces overhead to evaluate the predicate, the cost in terms of execution time is still negligible. It is worth noting that the slowdown is not very predictable. If we accidentally predicate a frequently executed basic block, the slowdown may be significant. On the other hand, the impact may be hardly noticeable for code that is not frequently executed.

The cost of unfolding frequently executed code (hot code) is in some cases too high to justify the adverse effect on the matching system. Therefore, we will avoid this type of code as well when introducing two-way opaque predicates. In this case, we will thus restrict ourselves to cold code: code that is neither hot nor frozen. The resulting cost is shown in Figure 4.9. The range after taking these restrictions into account is shown in Table 4.4.

The impact of the different transformations on the default matching system for $p = 0.5$ is shown in Figure 4.10. This shows that these transformations result in higher false negative rates than the unfolding transformations, while having a higher, but still moderate cost.

4.6.3 Control Flow Obfuscation

Control flow obfuscating transformations aim to make the control flow of the program less intelligible. This is typically achieved by either (i) adding redundant decision points, (ii) hiding the realizable paths within a large number of

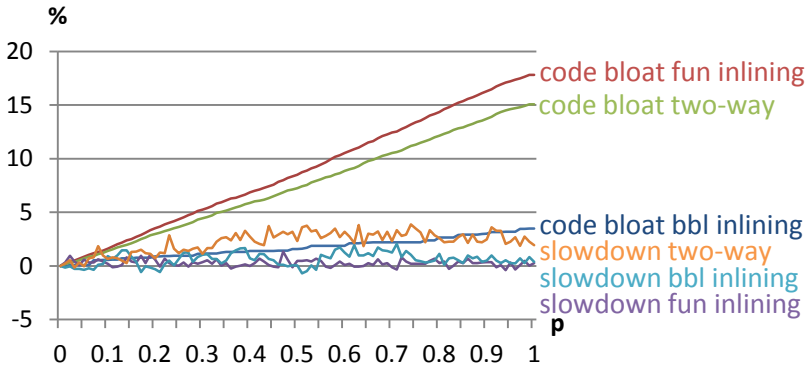
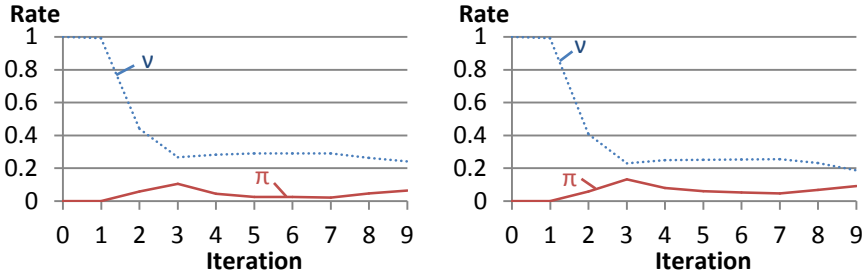
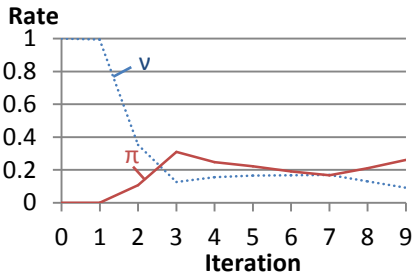


Figure 4.9: Code bloat and slowdown for the unfolding transformations



(a) Function inlining

(b) Basic block inlining



(c) Two-way opaque predicating

Figure 4.10: Evolution of the false positive and negative rate for the three unfolding transformations (p=0.5)

non-realizable paths, or (iii) invalidating common assumptions of static control flow graph construction algorithms.

Because they complicate and thus alter control flow, they are ready candidates to fool classifiers based on control flow information. Furthermore, their goal is to make it harder to understand the program and, therefore, they complicate intelligent tampering. Note that two-way opaque predicating can be seen as a form of control flow obfuscation as well.

Opaque Predicates

Our library of diversifying transformations contains one transformation based on true/false opaque predicates. A true (false) opaque predicate is a predicate that always evaluates to true (false) during execution, but for which this property is hard to detect for an attacker [Collberg 98b].

We can choose for each basic block whether or not to prepend it with a true or false opaque predicate. The target of the branch that cannot be executed can be chosen from the instructions in the function of the basic block. For each selected basic block, an opaque predicate can be chosen from a library of opaque predicates.

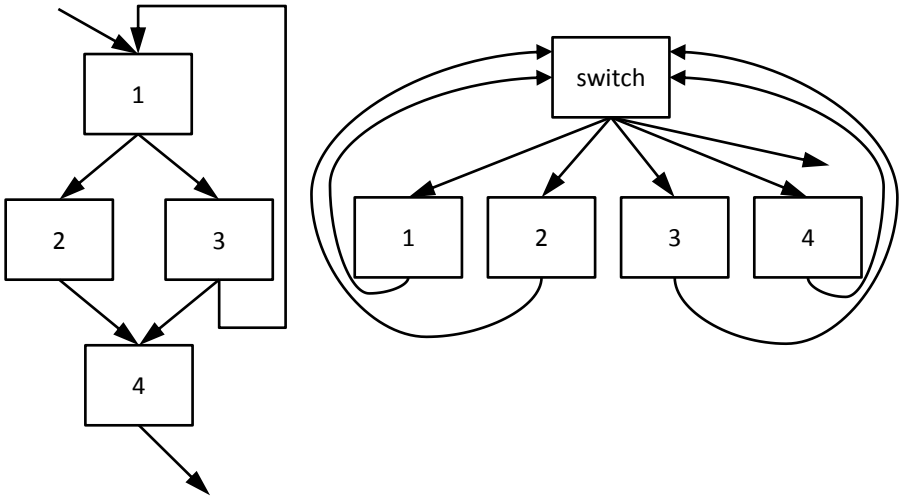
Control Flow Flattening

Control flow flattening significantly increases the number of paths represented by a control flow graph by replacing the original control flow graph by a new control flow graph where all original basic blocks have the same predecessor and successor [Wang 01]. The transformation is illustrated in Figure 4.11. Semantic equivalence is guaranteed by inserting a redirection variable guiding the execution.

We can choose for each function whether or not to apply control flow flattening to it. For each basic block in the selected function we can choose whether or not to redirect it.

Jump Redirection

We can choose for each jump instruction whether or not to redirect it through a branch function [Linn 03]. Branch functions are functions that do not return to the caller; instead control is transferred to a different address computed from the return address and offset passed to the branch function. Once a branch function is inserted, jumps can be transformed into calls to the branch function while arguments to correctly redirect control flow can be put on the stack. Figure 4.12(a) show an unconditional jump which is transformed into a call to the branch function in Figure 4.12(b).



(a) Original CFG (b) Flattened CFG

Figure 4.11: Control flow flattening

Evaluation

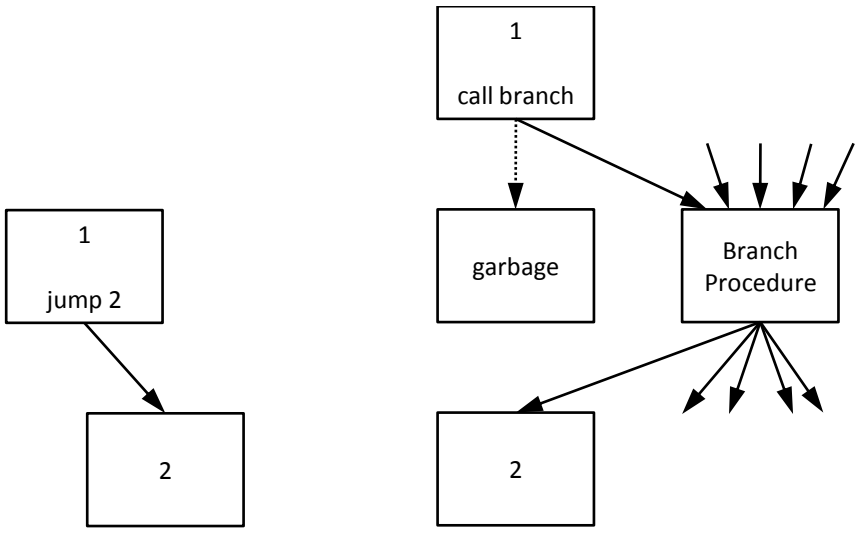
The number of candidates for the obfuscating transformations without restrictions is shown in Table 4.5. This does not take into account the choice between different opaque predicates, nor the choice on where to redirect the non-realizable path for opaque predicates, nor the selection of blocks to redirect for flattening.

As these transformations can incur a significant increase in code size and execution time, we have decided to limit them to cold code as well. The range after this restriction is also shown in Table 4.5.

The cost of these transformations after restrictions for $p \in [0, 0.6]$ is shown in Figure 4.13. We have chosen to impose an upper limit to the incurred overhead in code size and execution time of 10% per transformation. Therefore, we evaluate the impact of the transformations on the default matching system for $p = 0.2$ for opaque predicates, $p = 0.5$ for flattening and $p = 0.28$ for jump redirection. The results are shown in Figure 4.14.

4.6.4 Code Generation

We have grouped a number of diversifying transformations that are closely related to the backend phases of a compiler under the term code generation. These transformations are instruction selection, instruction scheduling, and code layout.



(a) Original code (b) Code using branch function

Figure 4.12: Jump redirection

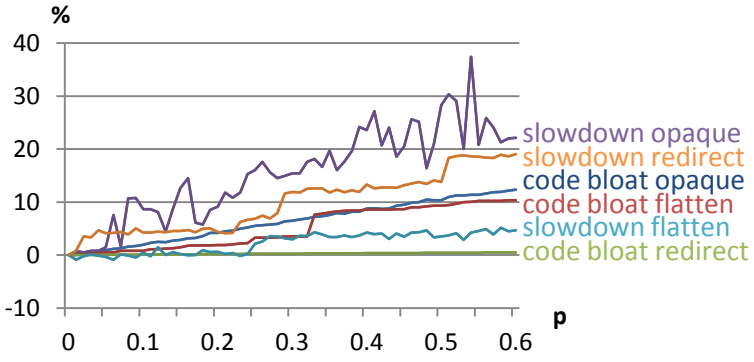
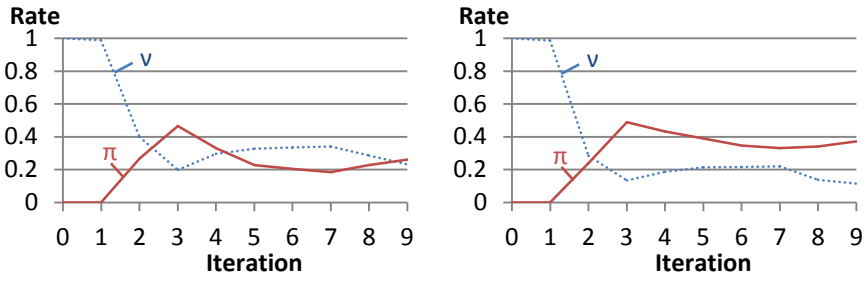
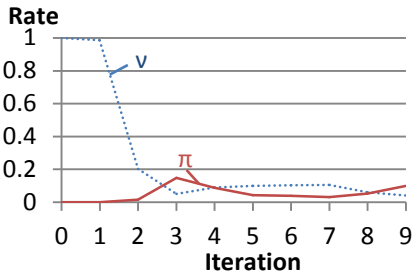


Figure 4.13: Code bloat and slowdown for the obfuscating transformations



(a) Opaque predicates (p=0.2)

(b) Control flow flattening (p=0.5)



(c) Jump redirection (p=0.28)

Figure 4.14: Evolution of the false positive and negative rate for control flow obfuscation

	No restrictions			With restrictions		
benchmark	opaque	flatten	jmp	opaque	flatten	jmp
400.perlbench	43978	1737	16831	9324	548	2560
401.bzip2	12583	502	5057	1354	81	205
403.gcc	111544	3510	37649	30335	1256	6383
429.mcf	11395	474	4768	1094	107	250
433.milc	13938	612	5330	2039	165	450
445.gobmk	31124	2712	10230	10493	981	2352
456.hmmer	16164	677	5933	2094	168	444
458.sjeng	14276	551	5914	1887	121	640
462.libquantum	12017	500	4903	1101	98	219
464.h264ref	25004	904	8223	5085	328	899
470.lbm	11491	474	4817	868	93	195
482.sphinx	15751	680	5498	3843	278	649

Table 4.5: Number of candidates for control flow obfuscation per benchmark

These different transformations are a compromise between an exhaustive search for semantically equivalent code sequences to perform a given operation and the time it takes to generate a semantically equivalent code sequence.

To explore the full range of semantically equivalent code sequences, we initially developed a tool that is capable of exhaustively generating all possible instruction sequences for the Intel Architecture – 32 bit (IA-32). This tool operates in a similar manner as the so-called superoptimizer [Massalin 87].

Its input consists of a code sequence, a set of output registers and a set of (scratch) registers whose value is no longer used after the sequence has been executed in a program. For all generated sequences, the tool checks whether they perform the same function as the original code sequence, by testing the output values for all possible input values. If the test succeeds an equivalent sequence is found.

Because of the halting problem, it is in general undecidable if a generated sequence will terminate. Hence the equivalence test can run forever. By restricting the set of instructions to the integer instructions, that do not include any control flow instructions, we can assure that each tested sequence terminates. But even then the number of potential equivalent sequences is still too large. To make the problem tractable, and to terminate the exhaustive generation within reasonable time, we further limit the immediate operands (constants encoded in an instruction) that can be used to $\{-1, 0, 1, 31\}$. Finally, we restrict the length of the generated sequences.

Even with these restrictions we can still find many equivalent sequences that perform realistic computations. For the operation $\texttt{ECX} = \max(\texttt{EAX}, \texttt{EDX})$,

e.g., our tool was able to find 433 different encodings of three instructions. Similarly, for the computation EAX= (EAX/2), 3708 equivalent sequences of 4 instructions were generated. Note that the tool did not find shorter sequences because of the limited list of immediates that does not contain 2.

It should be noted that these examples are no exception. Moreover, the number of alternatives is exponential in the number of instructions: if we have n instructions which we can divide into groups of i instructions of which each group has at least a alternatives, then combined we have at least $a^{n/i}$ alternatives. Furthermore, many additional alternatives arise when considering the larger fragment as a whole, in which instructions can be moved from one group to another.

While our tool thus shows great potential for generating diverse code fragments, it is too slow for a practical tool. Therefore, we split the generation of semantically equivalent code sequences into three parts: instruction selection, instruction scheduling and code layout.

Instruction Selection

We used the aforementioned tool to generate a database of equivalence classes for the instructions that occur most often in our suite of training programs. During this process, we imposed the additional restriction that equivalent instructions can only read/write locations that are read/written in the original instruction. However, if liveness analysis [Aho 86] determines that certain status flags are dead, we allow them to be overwritten as well. Finally, the set of immediates is expanded with the immediates used in the original instruction and the two's complement thereof.

We can choose for each operation in our database which of the semantically equivalent instructions to use.

Instruction Scheduling

Within each block there is some degree of freedom in the ordering of instruction, as two or more instructions that perform independent operations can be permuted.

This freedom is used for diversity by constructing a dependency graph of a block's instructions, in which dependent instructions are connected by directed edges. By iteratively removing instructions from this graph that do not depend on other instructions in it, a valid schedule can be determined. At each iteration, multiple instructions may be ready to be removed from the graph. They are, in other words, in the *ready set* [Aho 86] of instructions.

Whenever there are multiple instructions in the ready set, we can choose which one to schedule next.

Code Layout

Once the order of instructions in a basic block is determined, we still need to determine the order of basic blocks in the program. There is a lot of freedom in choosing this order. Only fall-through paths need to be respected. Fall-through paths connect basic blocks between which control can be transferred without explicit indication of the target of the transfer. Examples include: the basic block at which the execution continues when a callee returns, the location to which control is transferred when a conditional jump is not taken.

Firstly, in many cases we can choose which of the targets of a conditional jump to select as the fall-through block as for every conditional jump instruction, a jump instruction with the inverted condition exists in our target architecture (x86). Then, all basic blocks connected by fall-through paths are chained together.

Secondly, we can freely choose the order of the generated chains. We can then iteratively select the next chain to place in the final layout of the program.

Evaluation

The number of choices available during code generation is huge. To facilitate comparing the ranges to the ranges of the previously evaluated transformations, we have normalized them to choices between two alternatives. All of the ranges shown hitherto indicate the choice between applying the transformations or not applying them, i.e., two alternatives.

For example, if we have to choose between 5 alternatives, we will report this as $\log_2(5)$ choices between two options. As a result, the range is still given by 2^n for n in Table 4.6.

The scheduling and code layout transformations have little effect on the classifiers discussed in Chapter 3. Yet, they do defend against text-based comparisons. Furthermore, the choice of classifiers is based on the assumption that code will be reordered. If we could assume that this is not the case, one could build classifiers to exploit this knowledge.

The cost of choosing a random schedule rather than the compiler-generated schedule is negligible in terms of code size and execution time, as we are working on a superscalar architecture with out-of-order execution. The cost of choosing a random layout, rather than a more logical order (group related code to minimize jump offsets and maximize spatial locality) is negligible for the mcf benchmark in terms of execution time and about 7% in terms of code size. In a random layout, many more (jump) offsets will have to be encoded in four bytes as opposed to one byte. Code layout does have a significant impact on the execution time for some of the other benchmarks. This is due to caching effects which become more prevalent for programs with a larger kernel.

benchmark	instruction selection	scheduling	code layout
400.perlbench	68530.25	42309.83	253725.55
401.bzip2	24386.30	16552.32	64176.33
403.gcc	170790.70	94904.96	607946.06
429.mcf	21936.82	13428.83	59641.90
433.milc	24796.76	17232.51	70292.00
445.gobmk	53826.39	49332.87	169634.26
456.hmmer	28219.20	18922.05	79437.27
458.sjeng	27409.95	17503.91	75258.28
462.libquantum	22840.17	14830.94	61940.67
464.h264ref	43884.79	43460.11	114463.68
470.lbm	22034.62	13704.27	60322.90
482.sphinx	27334.81	18663.84	75663.56

Table 4.6: Number of choices for code generation per benchmark, normalized to choices between two options for ease of comparison

To evaluate the impact of selecting a random instruction rather than the shortest, or fastest alternative, we have applied it with different probabilities: replace instructions by equivalent ones in 0-100% of the cases where it is possible. This cost is illustrated in Figure 4.15.

We can see that the code size increase is linear and fairly predictable. Code size increase occurs when an instruction in the original program is replaced by a semantically equivalent, but longer instruction.

The slowdown is less predictable, although we can observe a slight increase when the transformation is applied more times. We assume that cache effects resulting from different code size and layout are responsible for this variation.

The impact of the diversity introduced during code generation on the default matching system is shown in Figure 4.16. These transformations have very little effect on the matching system at hand. Yet, as already mentioned, any diversity system should apply reordering, directly or indirectly, to thwart matching systems which take the location of code in the program into account.

4.7 Evaluation

In the previous measurements, we have used only two different seeds for the pseudorandom number generator: 1 and 2. This may have lead to accidentally poor or accidentally good results. We will first experimentally verify that the thus obtained results are representative for other (combinations of) seeds as well.

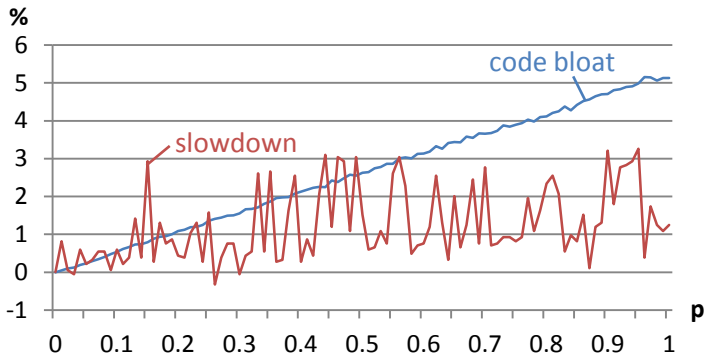


Figure 4.15: Code bloat and slowdown for instruction selection

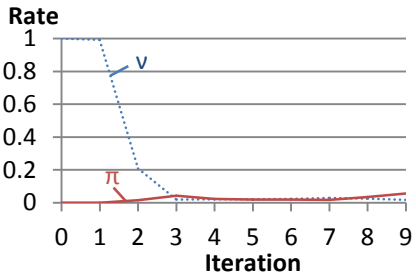


Figure 4.16: Evolution of the false positive and negative rate for the code generating transformations

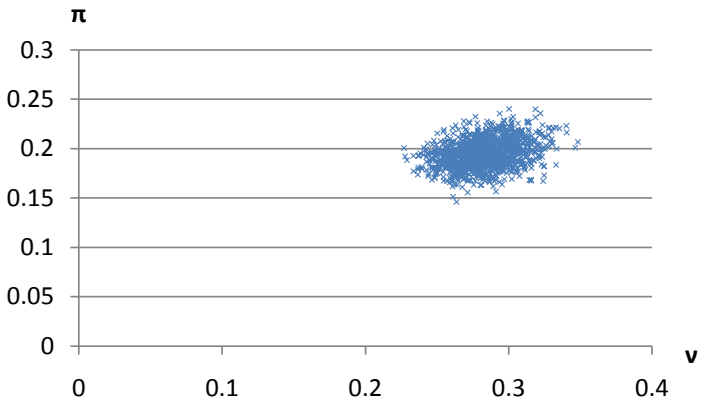


Figure 4.17: False positive and false negative rates for different combinations of seeds for the mcf benchmark

Next, we will look at the effect of combining different diversifying transformations on the default matching system. We will also evaluate the impact of the combined transformations on the different individual classifiers for different thresholds. Then, we will study how portable the results are to the other programs in the SPEC CPU2006 benchmark suite.

Finally, we will evaluate the transformations based on code generation within a different context: steganography.

4.7.1 Representativeness of the Seeds

To experimentally verify that the resulting false positive and false negative rates are relatively independent of the chosen seeds, we have set up the following experiment: we have generated 50 versions for a given transformation and evaluated the impact on the default matching system for any two versions generated. This leads to $\binom{50}{2} = 1225$ experiments.

We have chosen to apply this experiment to the transformation based on opaque predicates, as this transformation has one of the higher impacts on the matching system and the experiments finish within a time frame acceptable for this number of experiments. Other transformations have a smaller impact on the matching system, or significantly delay the matching process. The jump redirection and control flow flattening, for example, increase the sets of neighboring blocks and thereby slow down the classifier based on control flow.

We have plotted the obtained false positive and false negative rates of the default matching system in Figure 4.17 for the different experiments.

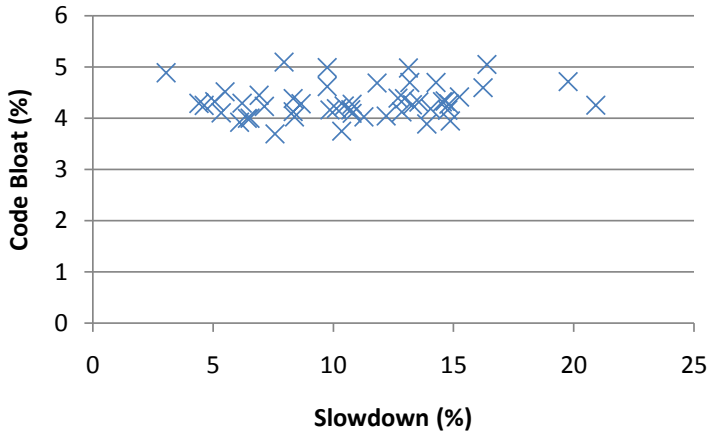


Figure 4.18: Code bloat and slowdown for the different seeds for the mcf benchmark

The results show that the errors are relatively constant for every evaluated combination of seeds. The average false negative rate is 0.28 with a standard deviation of only 0.0182. The average false positive rate is 0.1953 with a standard deviation of only 0.0109. We are confident that similar observations can be made for the entire range of random seeds. In the current implementation, these are limited to 2^{32} values, enabling us to generate as many versions with a similar degree of diversity.

The impact on the execution time is less predictable, as shown in Figure 4.18 due to the variance in the execution count of the transformed code, despite the limitation to cold code, with an average of 10.81% and a standard deviation of 4.03. If this proves to be a major issue in practical settings, it could be partially solved through better profiling and a more fine-grained distinction between execution counts (as opposed to only three possibilities: frozen, cold and hot). The impact on the code size, on the other hand, is relatively predicable, with an average of 4.33% and a standard deviation of 0.32.

4.7.2 Combining Transformations

We have combined the different transformations to evaluate the impact on the matching system. The settings for the different transformations are the same as the settings used for the individual transformations. They are applied in the order they are treated above. Furthermore, we have made sure that the unfolding transformations cannot undo the folding transformations.

The impact on the default matching system is shown in Figure 4.19. The matching system fails to identify 76% of the required matches. About 58%

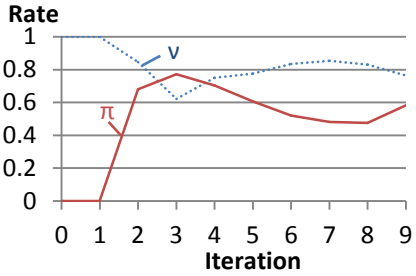


Figure 4.19: Evolution of the false positive and negative rate for the combined transformations for the mcf benchmark

of the reported matches are between unrelated code fragments. With these settings, the slowdown is around 54%, while the code bloat is around 70%. This shows that we are able to fool the matching system to a large extent, however at a relatively high cost.

4.7.3 Receiver Operating Characteristic Curves

We will evaluate the impact of the combined transformations on the individual classifiers through ROC curves. ROC curves have been used extensively in signal detection theory and are graphical plots of the True Positive Rate (TPR) versus the False Positive Rate (FPR) as the threshold of the classifier is varied.

Using our notation, the true positive rate equals $1 - \nu$ and the false positive rate is ϕ. An example of an ROC curve is shown in Figure 4.20. A point on the line of no discrimination indicates that the classifier performs no better than a random guess. The optimal result is the point at coordinate $(0, 1)$.

As can be derived from Figure 4.21, the performance of the classifier based on instruction syntax is much better than a random guess. This indicates that the diversity is far from perfect with respect to this classifier. However, it should be noted that the false positive rate (ϕ) is expressed as the fraction of the worst-case number of false positives. Hence, even a small false positive rate represents many false positives.

The classifier based on data has a good performance when we only take code fragments into account for which it does have an opinion. We have also plotted the results when code fragments for which the classifier has no opinion are marked as unrelated. This is shown in Figure 4.22.

As we can see from Figure 4.23 and Figure 4.24, the classifiers based on execution count and first execution time have a fairly high false positive rate. This shows that these classifiers cannot easily identify unrelated code fragments. This is often referred to as a poor specificity.

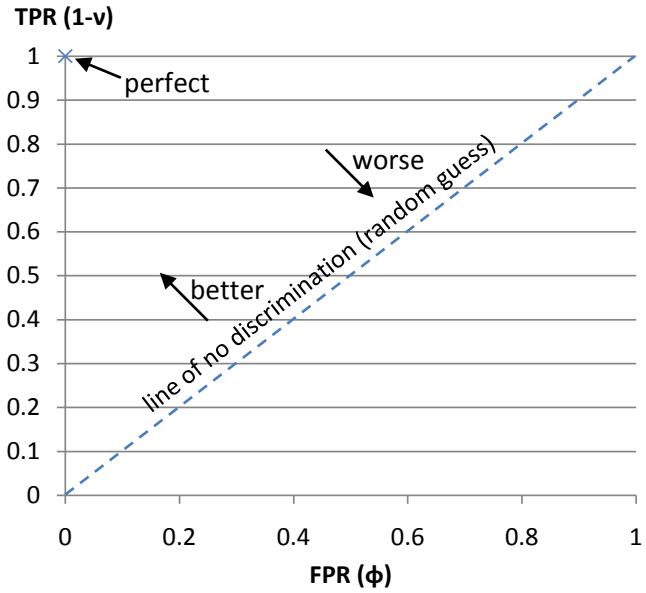


Figure 4.20: An example of an ROC curve

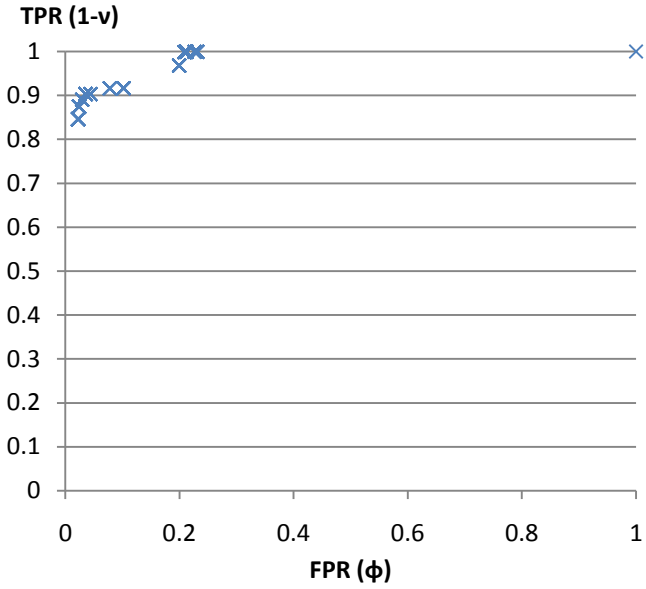


Figure 4.21: ROC curve for the classifier based on instruction syntax

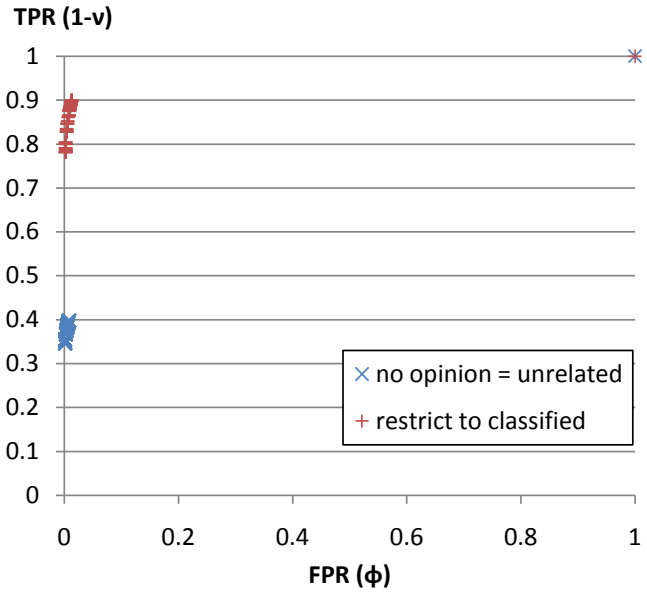


Figure 4.22: ROC curve for the classifier based on data

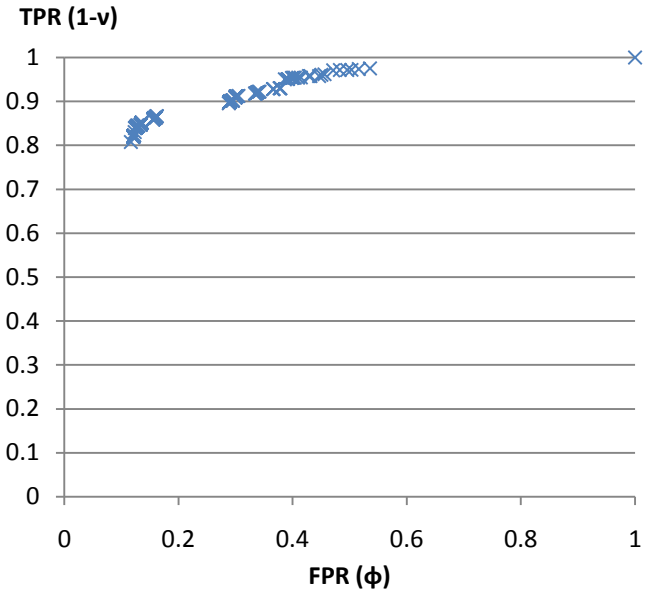


Figure 4.23: ROC curve for the classifier based on execution count

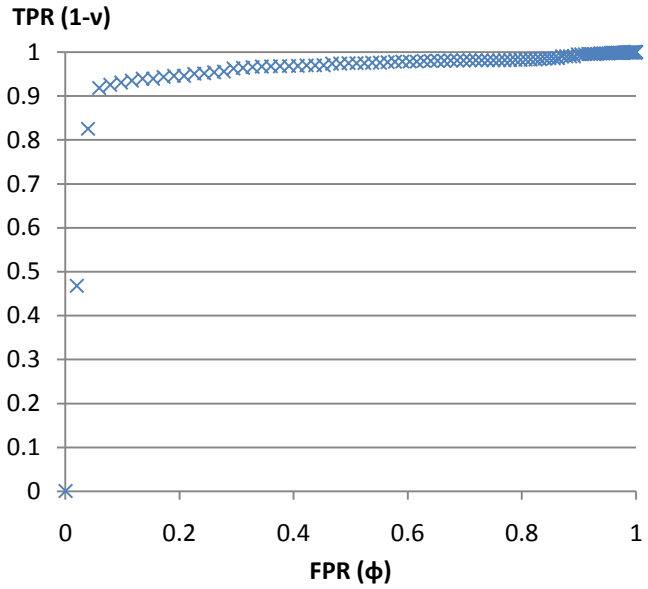


Figure 4.24: ROC curve for the classifier based on first execution time

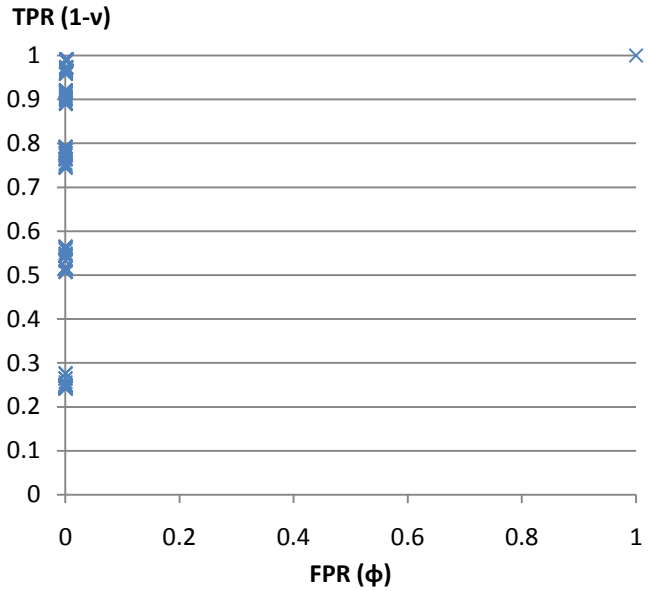


Figure 4.25: ROC curve for the classifier based on control flow

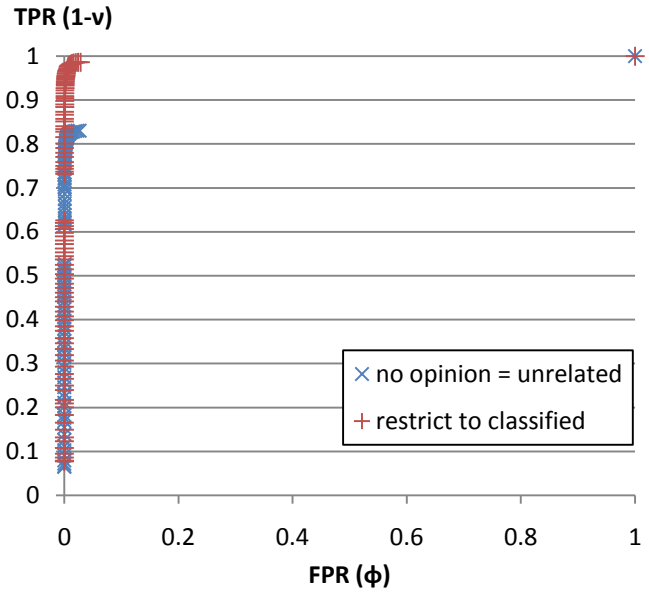


Figure 4.26: ROC curve for the classifier based on data flow

The classifiers based on control flow and data flow on the other hand seem to perform very well (Figure 4.25 and Figure 4.26). However, these classifiers require an existing mapping. The reported results are under the assumption that all pairs of code fragments, except for the pair under consideration, have been classified correctly. As such, these results indicate a high potential for the classifiers based on control flow and data flow to extend or filter a sufficiently accurate mapping.

4.7.4 Representativeness of the Benchmark

We have applied the diversity system defined by the settings shown in Table 4.7 on the C programs of the SPEC CPU2006 benchmark suite. Note that these settings are different from the settings used earlier for the mcf benchmark. We have selected different, less powerful settings to limit the slowdown incurred by larger sets of neighboring blocks, e.g., due to jump redirection and control flow flattening.

The resulting false positive and false negative from the default matching system are shown in Figure 4.27. The average false negative rate is 0.48, with a standard deviation of only 0.0809. The average false positive rate is 0.24, with a standard deviation of 0.0225.

Transformation	p
Function factoring	0.5
Epilogue factoring	0.5
Basic block factoring	0.5
Function inlining	0.1
Two-way opaque predicating	0.1
Basic block inlining	0.1
Control flow flattening	0.1
Jump redirection	0.1
Opaque predicating	0.1
Instruction selection	1
Instruction scheduling	1

Table 4.7: Settings of the diversity system

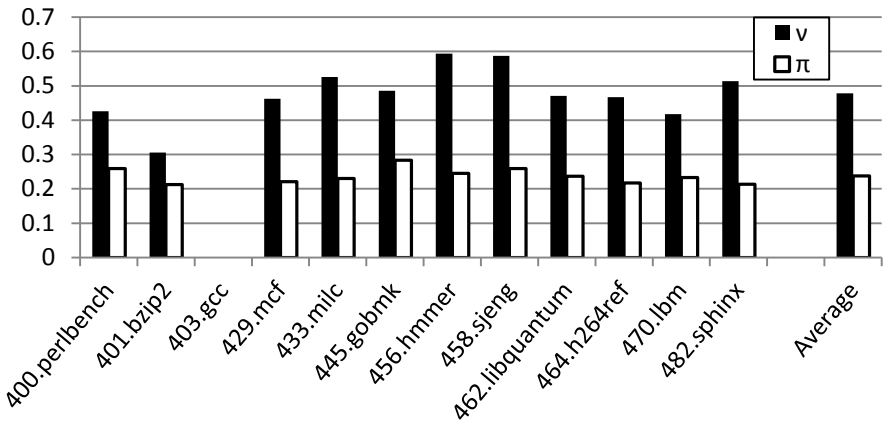


Figure 4.27: False positive and false negative rate after the last iteration of the default matching system

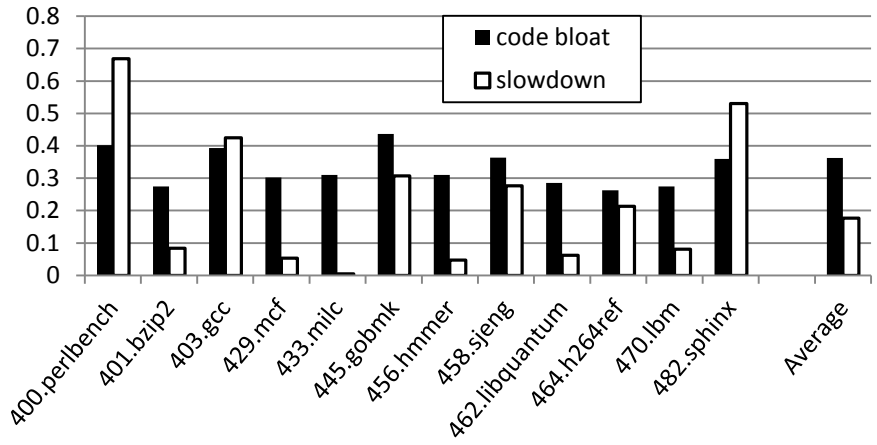


Figure 4.28: Code bloat and slowdown for the different benchmarks

The cost of applying this diversity system is a 36.2% code increase (with a standard deviation of 0.0577) and a slowdown of 17.7% (with a standard deviation of 0.2165).

4.7.5 Steganography – Histiæus

The first reported occurrence of steganography is due to Herodotus. He tells of Histiæus, who shaved the head of his most trusted slave, tattooed a message on his head, and then waited for his hair to grow back. The slave was then sent to Aristagoras, who was instructed to shave the slave's head again and read the message.

Many other physical objects have since been used as cover objects, e.g., earrings, written documents, and music scores. Digital steganography has mainly been applied to media, such as images, sound and video.

Steganography in the context of programs has, to the best of our knowledge, only been addressed by Hydan [El-Khalil 04]. We will apply the transformations based on code generation (instruction selection, instruction scheduling and code layout) within the context of steganography.

The Prisoners' Problem

We will follow Simmons' [Simmons 84] classic model, a.k.a. *the prisoners' problem* for invisible communication. Alice and Bob are two prisoners in different cells. Wendy, the warden, arbitrates all communication between them,

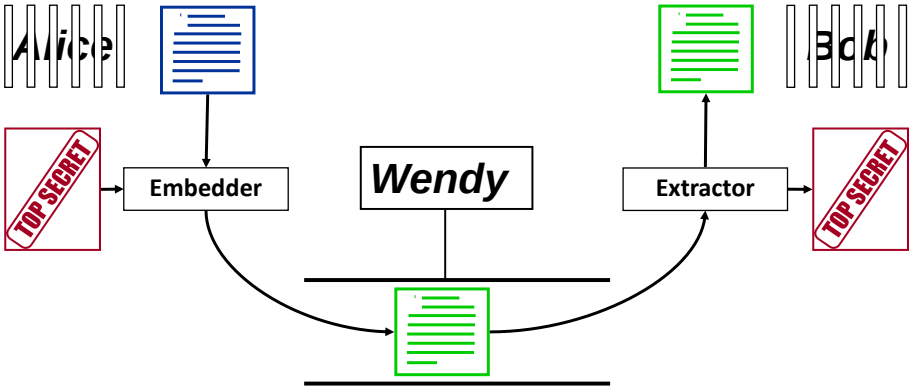


Figure 4.29: The prisoners' problem

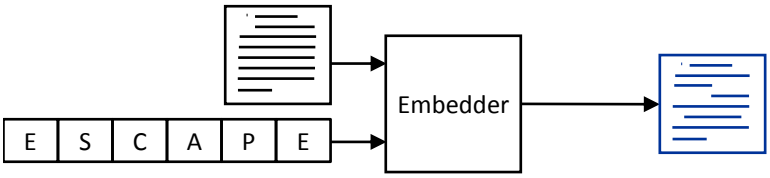
and will not let them communicate through encryption or suspicious communication. Both prisoners therefore need to communicate invisibly about their escape plan.

Furthermore, we will assume that the mechanism in use is known to the warden (Kerkhoffs' principle [Kerkhoffs 83]). Hence its security must depend solely on a secret key that Alice and Bob managed to share, possibly before their imprisonment.

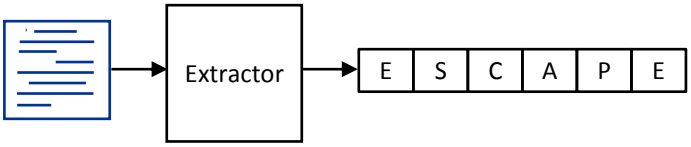
The general principle of steganography is as follows (see Figure 4.29). To share a secret message with Bob, Alice randomly chooses a harmless message, called a cover object c, which can be transmitted to Bob without raising suspicion. The secret message m is then embedded in the cover object using the secret key k, resulting in a stego object s. This is to be done in such a way that Wendy, knowing only the apparently harmless message s, cannot detect the presence of the secret. Alice then transmits s to Bob via Wendy. Bob can reconstruct m since he knows the embedding method and has access to the key k. It should not be necessary for Bob to know the original cover c. The security of invisible communication lies mainly in the inability to distinguish cover objects from stego objects. The task of Wendy can be formalized as a statistical hypothesis testing problem, for which she defines a test function on objects (of the set O) $f : O \rightarrow \{0, 1\}$:

$$f(o) = \begin{cases} 1 & \text{if } o \text{ contains a secret message} \\ 0 & \text{otherwise} \end{cases} .$$

This function can make two types of errors: detect a hidden message when there is none (false positive) and not detect the existence of a hidden message when there is one (false negative).



(a) The secret message is used as nonce



(b) Extracting the secret message from the cover object

Figure 4.30: Using a diversity system for steganography

We furthermore assume that the warden is passive, i.e., she will not modify the object, but only classify it. This is generally accepted in steganography [Anderson 98]. On the other hand, watermarking and fingerprinting (see Section1.2.2) typically assume an active warden.

Fitness of Programs as Cover Objects

While changing a single bit in a program can cause it to fail, this does not imply a lack of redundancy for the purpose of steganography. Instead the specific characteristics of software indeed result in many choices. So far, we have used these choices to create many different versions. We can use the same choices to embed a secret message in the program. The nonce is then replaced by the secret message, resulting in a single copy with an embedded message. This is illustrated in Figure 4.30.

For each of the choices between alternatives, a number of bits can be encoded in the program. If there are n equivalent programs because of some type of choice, the number of bits that can be encoded can be computed as follows.

As $n \geq 2^{\lfloor \log_2(n) \rfloor}$, it is clear that at least $\lfloor \log_2(n) \rfloor$ bits can be encoded: it suffices to assign an integer to each alternative, and to take that alternative whose (binary) number corresponds to the bit string to be encoded. This simple approach may result in a significant decrease in encoding capabilities however: if $\log_2(n) \notin \mathbb{N}$ for large n, many alternatives may not correspond to an encodable bit string.

A more efficient scheme is as follows: if $\log_2(n) \notin \mathbb{N}$, then $\lfloor \log_2(n) \rfloor = \lceil \log_2(n) - 1 \rceil$. We can thus always embed $\lceil \log_2(n) - 1 \rceil$ bits. If we associate each of the remaining $n - 2^{\lceil \log_2(n) - 1 \rceil}$ alternatives with one of the $2^{\lceil \log_2(n) - 1 \rceil}$

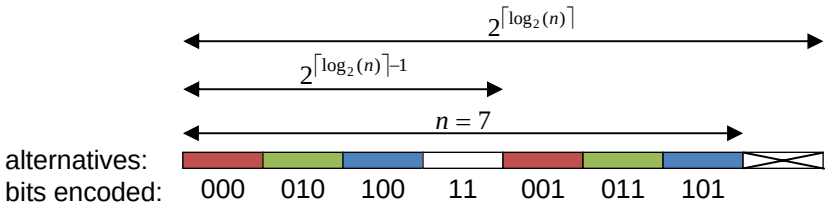


Figure 4.31: Encoding bits in the choice between 7 alternatives

already used ones, we can embed an additional bit by allowing the embedder to choose between one of the two associated alternatives, as illustrated for $n = 7$ in Figure 4.31. Therefore, we can embed an extra bit in $n - 2^{\lceil \log_2(n)-1 \rceil}$ of the $2^{\lceil \log_2(n)-1 \rceil}$ possibilities for the next $\lceil \log_2(n) - 1 \rceil$ bits.

If the embedded message is encrypted with the secret key k, all bit strings to be embedded have equal probability, and hence the average number of bits that can be encoded in the choice out of n valid alternatives is given by

$$b(n) = \lceil \log_2(n) - 1 \rceil + \frac{n - 2^{\lceil \log_2(n)-1 \rceil}}{2^{\lceil \log_2(n)-1 \rceil}} \quad . \tag{4.1}$$

One can easily verify that equation (4.1) also holds if $\log_2(n) \in \mathbb{N}$. As illustrated in Figure 4.32, the average number of embeddable bits is a lot closer to $\log_2(n)$ in this scheme.

Interactions Between the Techniques

The techniques (instruction selection, instruction scheduling and code layout) are not completely orthogonal. In order to combine them successfully, a couple of issues need to be addressed.

First, it is worth noting that the number of bits that can be encoded in instruction selection is dependent on the chosen ordering of instructions in the basic block, and vice versa. When the ordering changes, liveness ranges change, and hence the condition flags and scratch registers that may be changed by equivalent instructions also change.

For the same reason, instruction selection influences the order in which an embedder or extractor will generate equivalent orderings, and hence how specific bit sequences are encoded in the ordering. Vice versa, if scheduling is applied first, it influences the order in which equivalent instructions are generated.

Moreover, if the embedder first encodes bits in the instruction selection of the instructions in their original order in the program, and subsequently reorders the instructions, the extractor does not know the order in which the information embedded in the instruction selection needs to be extracted. Clearly,

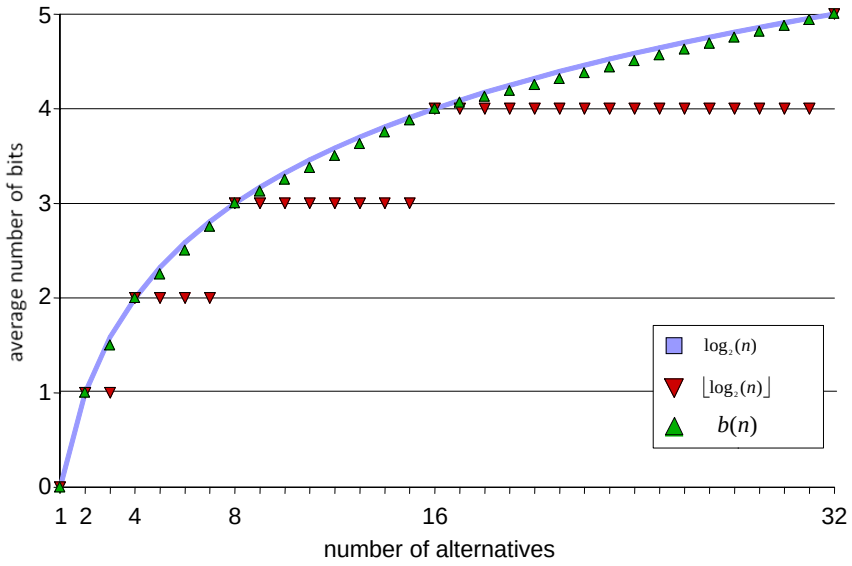


Figure 4.32: The average number of embeddable bits for a choice between alternatives

the extractor and the embedder need to start from the same dependency graph in order for the extractor to obtain the correct embedded information.

Before the embedding and the extraction, all basic blocks in a program should therefore be transformed into a canonical form, in which both the instruction selection and scheduling are predetermined. This is illustrated in Figure 4.33.

Practical Considerations for Extracting an Embedded Message

In order to extract embedded information from a program, an extractor needs to identify the basic blocks, and the extractor needs to pinpoint relocated operands, since these should be neglected for the ordering of chains.

The necessary relocation information is available at the embedding phase, as the embedding is done at link time, when the whole program is first available. However, this information is lost in the resulting program.

Fortunately most of the necessary information can be derived from a static analysis of the program itself. As a consequence, we only need to communicate the discrepancy between the derived information and the actual information to the decoder. To do so, we can store this information in the first instructions of the resulting program, without taking liveness information into

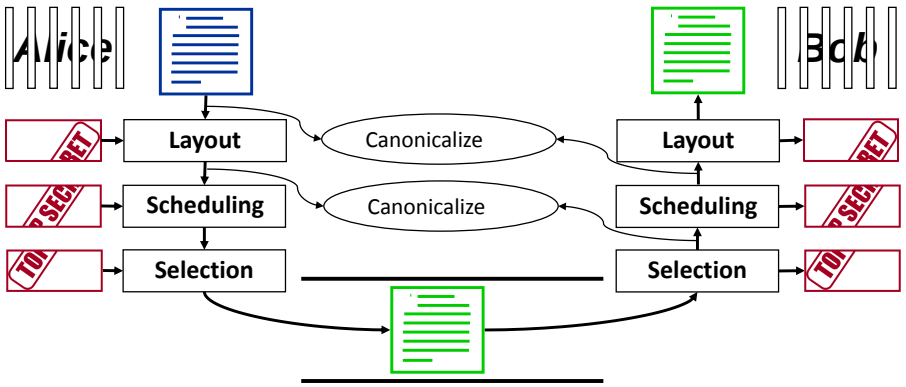


Figure 4.33: Canonicalization ensures that the encoding and decoding phases start from the same information

account. This is the only option since the decoder cannot identify basic blocks or chains and it cannot compute liveness information at this point.

Code Transformation Signatures

While the encoding rate achieved by the discussed techniques is fairly high, its security is too low. The reason is that the techniques introduce very unusual code that will arouse suspicion of the warden. Consider, e.g., the equivalent code sequences in Figure 4.34. Anyone somewhat familiar with assembly code will agree that the likelihood of a compiler generating the code on the right is extremely low. But this code is present in programs that have been put through Hydan or on which our techniques have been applied (without countermeasures). In short, the application of our tool has left an obvious signature.

We have identified four types of code transformation signatures that may reveal the presence of an embedded message. Firstly, there are a number of instructions which do not appear at all in programs created by a typical tool chain. Secondly, in the case that multiple equivalent instructions, say n, are used in practice, their relative frequency is not $1/n$ in regular programs. However, it will be when using instruction selection to embed messages (if they are encrypted beforehand). Thirdly, different tool chains may generate different schedules for identical basic blocks, because they use different heuristics, target a different micro-architecture, etc. However, the same tool chain will show little diversity in scheduling identical basic blocks. Finally, the average jump offset will typically be small. This is, a.o., the result of most jumps being intraprocedural. If code layout is random, the average jump offset will be suspiciously large.

```
55          push EBP          |   55          push EBP
89 e5       mov  ESP,EBP      |   89 e5       mov  ESP,EBP
83 ec 08    sub  0x8,ESP      |   83 c4 f8    add  0xfffffff8,ESP
```

Figure 4.34: Two equivalent code sequences.

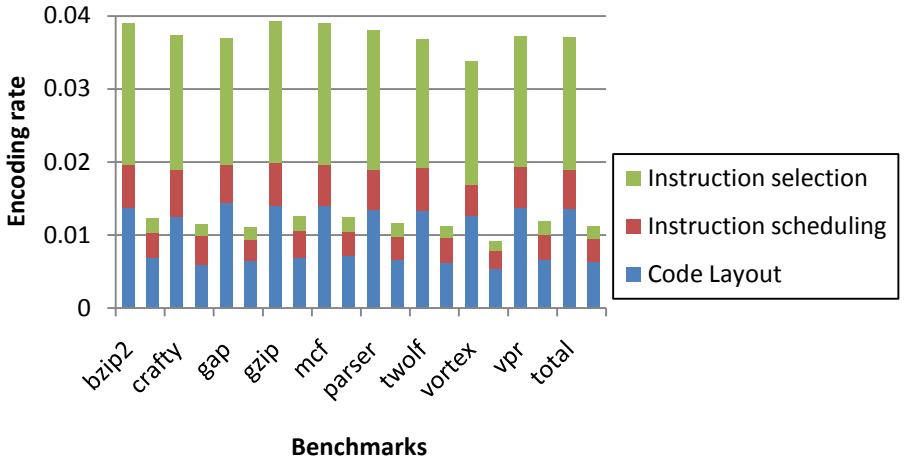


Figure 4.35: Encoding rate before (left) and after (right) countermeasures for steganalysis

Using statistical models built upon a collection of regular programs, we have restrained the freedom of the different transformations to avoid being detected by these code transformation signatures.

Results

We have applied the techniques on 9 integer programs of the SPEC CPU2000 benchmark programs to embed and extract the plaintext of "King Lear" by William Shakespeare. The programs were compiled with GCC 3.2.2 and statically linked to glibc 2.3.2 for Linux. For each benchmark, the embedding and extraction took less than a minute on a 2.8GHz Pentium IV.

The obtained encoding rates are presented in Figure 4.35. The distribution over the different techniques is also indicated. We achieve an encoding rate between $1/29.6$ and $1/25.49$ and a total encoding rate of $1/26.96$ before countermeasures, four times the encoding rate of the previous prototype tool Hydan $(1/110)$.

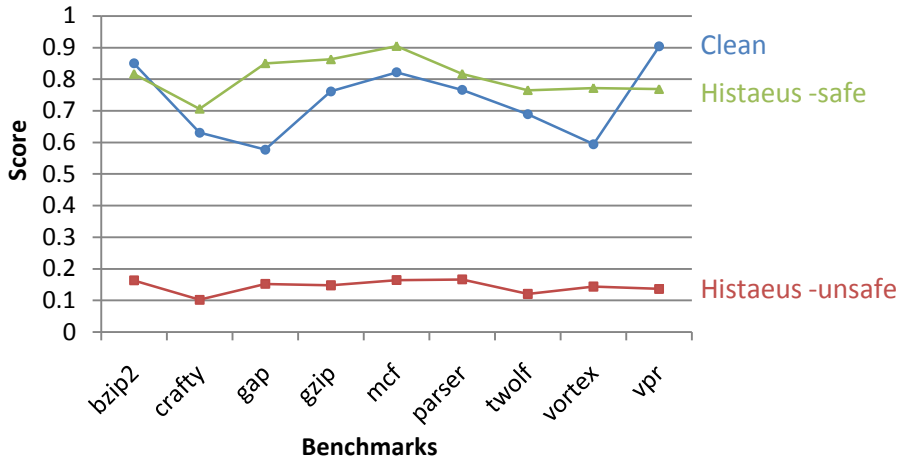


Figure 4.36: Code transformation signature: unusual relative frequencies of instructions

Without countermeasures, neither our tool nor Hydan, as noted by its authors, is stealthy. Most importantly, their encoding rate achieved through instruction selection is achieved by and large by choosing very unusual instructions. However, as the clean instruction distribution is uniform across programs [El-Khalil 04], little information can be stealthily embedded this way. When we take countermeasures to address this problem, only 9% of the encoding rate due to instruction selection remains, as can be seen in Figure 4.35. This clearly illustrates the usual trade-off between data rate and stealth.

The presence of code transformation signatures is illustrated in Figure 4.36. These results have been obtained as follows. Suppose that there are n alternatives for some instruction, whose relative frequencies in a program p are represented by $f(p) = [f_1(p),\ ...\ f_n(p)]$. We then compute the Euclidean distance from $f(p)$ to $f(P)$, in which P is large set of reference programs.

We then postulate that this distance follows a normal distribution, of which the parameters μ and σ^2 can be estimated by computing the average and variance of the distance in the set of reference programs.

The score of a program p is the probability of observing a distance larger than the observed distance between $f(p)$ and $f(P)$. A low likelihood indicates a high probability that a secret message is present. The reliability of this test is increased by averaging the likelihoods of a number of sets of alternative instructions. The classification could then classify programs with a score below, e.g., 0.5 as suspect. These results confirm that unlimited instruction selection freedom indeed results in big, easily detectable discrepancies from clean code.

Fortunately, there is a much greater variation in schedules across programs and 47% of the encoding rate due to instruction scheduling can be safely retained. Furthermore, as code layout is largely determined by the source code, a piece of information that is no longer available to the warden, 59% of the encoding rate due to code layout can be safely retained.

Combined, we achieve a stealthy encoding rate ranging from $1/108.59$ to $1/80.1$.This is still higher than the unsafe rate of Hydan.

5 Advanced Transformations

In the previous chapter, we have studied the effectiveness of a number of existing transformations from different domains within the context of software diversity. In this chapter, we will look at two techniques which have been developed from the ground up with diversified tamper resistance in mind.

The main benefit of the first of these techniques, self-modifying code, is that it undermines the commonly made assumption that code is constant. As such, many tools are not able to deal correctly with this type of code. Other tools can work conservatively with this type of code, but are optimized for the common case, i.e., constant code. As a result, they experience a significant delay because of self-modifying code.

The goal of the second technique, virtualization, is to open up a whole new range of possibilities. Virtualization gives us the freedom to design our own Instruction Set Architecture (ISA). This results in a large number of choices and thus a lot of room for diversity. Furthermore, it allows us to abandon traditional execution models, which may lead to increased tamper resistance.

5.1 Self-modifying Code

Self-modifying code has a long history of hiding program internals. It was used to hide copy protection instructions in 1980s MS DOS based games. The floppy disk drive access instruction `'int 0x13'` would not appear in the program's image but it would be written into the program's memory image after the program started executing.

While hiding the internals of a program can be used to protect the intellectual property contained within or protected by software, it has been applied for less righteous causes as well. Viruses, for example, try to hide their malicious intent through the use of self-modifying code [Leprosy 90].

Self-modifying code is very well suited for these applications as it is assumed to be one of the main problems in reverse engineering [Cifuentes 95]. Because self-modifying code is so hard to understand, maintain and debug, it is rarely used nowadays. As a result, many analyses and tools make the assumption that code is not self-modifying, i.e., constant. Note that we distinguish self-modifying code from run-time generated code as used in, e.g., a Java Virtual Machine.

Our goal is to leverage the known complications of self-modifying code to increase the tamper resistance of individual copies, and, by making the transformations parameterizable, increase the diversity between versions. The presence of self-modifying code can delay the collection of information used by the classifiers and furthermore forces them not to assume that the code is constant.

To make this possible, we need a representation that enables us to turn constant code into self-modifying code and to analyze, transform and linearize the resulting self-modifying code. Fortunately, we are at an advantageous position, as our starting point is a known, constant-code program, whereas a matching system starts from unknown, possibly self-modifying program.

The representation often used for traditional code, which neither reads nor writes itself, is the control flow graph. Its main benefit is that it represents a superset of all executions. As such, it allows analyses to reason about every possible run-time behavior of the program. Furthermore, it is well understood how a control flow graph can be constructed, how it can be transformed and how it can be linearized into an executable program.

Until now, there was no analogous representation for self-modifying code. Existing approaches are often ad hoc and usually resort to overly conservative assumptions: a region of self-modifying code is considered to be a black box about which little is known and to which no further changes can be made.

We will discuss why the basic concept of the control flow graph is inadequate to deal with self-modifying code and introduce a number of extensions which can overcome this limitation. These extensions are: (i) a data structure keeps track of the possible states of the program, (ii) an edge can be conditional on the state of the target memory locations, and (iii) an instruction uses the memory locations in which it resides.

We refer to a control flow graph augmented with these extensions as a state-enhanced control flow graph. These extensions ensure that we no longer have to artificially assume that code is constant. In fact, existing data analyses can now readily be applied on code, as desired in the model of the stored-program computer. Furthermore, we will discuss how the state-enhanced control flow

graph allows for the transformation of self-modifying code and how it can be linearized into an executable program.

The Running Example For our example, we introduce a simple and limited instruction set which is loosely based on the 80x86. For the sake of brevity, the addresses and immediates are assumed to be 1 byte. It is summarized below:

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set value of byte *to* to *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*

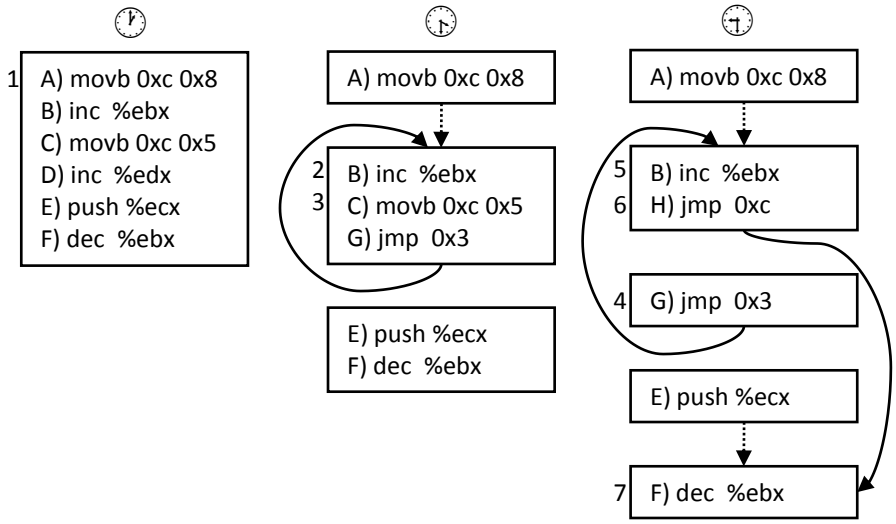
As a running example, we have chosen to hide one of the simplest operations. The linear disassembly of the obfuscated version is as follows:

Address	Assembly	Binary
`0x0`	`movb 0xc 0x8`	`c6 0c 08`
`0x3`	`inc %ebx`	`40 01`
`0x5`	`movb 0xc 0x5`	`c6 0c 05`
`0x8`	`inc %edx`	`40 03`
`0xa`	`push %ecx`	`ff 02`
`0xc`	`dec %ebx`	`48 01`

If we would perform traditional CFG construction on this code, we would obtain a single basic block as shown in Figure 5.1(a). If we step through the program however, we can observe that instruction `A` changes instruction `D` into instruction `G`, resulting in a new CFG as shown in part (b). Next instruction B is executed, followed by instruction `C` which changes itself into jump instruction `H` (c). Then, instruction `G` transfers control back to `B` after which `H` and `F` are executed. The only possible trace therefore is `A,B,C,G,B,H,F`. While not apparent at first sight, we can now see that these instructions could be replaced by a single instruction: `inc %ebx`.

5.1.1 The State Enhanced Control Flow Graph (SE-CFG)

CFGs have since long been used to discover the hierarchical flow of control and for data flow analysis to determine global information about the manipulation of data [Muchnick 97]. They have proved to be a very useful representation enabling the analysis and transformation of code. Given the vast amount of research that has gone into the development of analyses on and transformations



(a) before execution (b) after the first write instruction A (c) after the second write instruction C

Figure 5.1: Traditional CFG construction. The numbers indicate the actual execution order of instructions

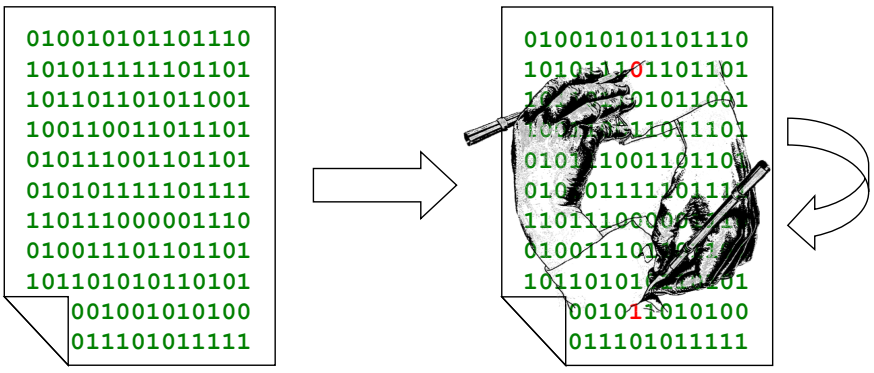


Figure 5.2: The SE-CFG enables the transformation of constant code into self-modifying code and the analysis and transformation of the thus obtained self-modifying code

of this program representation, we are eager to reuse the knowledge resulting from this research.

A Control Flow Graph for Self-Modifying Code

One of the reasons a CFG is so useful is that it represents a superset of all the possible executions that may occur at run time. As a result, many analyses rely on this representation to reason about every possible behavior of the program. Unfortunately, traditional CFG construction algorithms fail in the presence of self-modifying code. If they are applied on our running example at different moments in time, we obtain the three CFGs shown in Figure 5.1. However, none of these CFGs allows for both a conservative and accurate analysis of the code.

We can illustrate this by applying unreachable code elimination on these CFGs. This simple analysis removes every basic block that cannot be reached from the entry block. If it is applied on Figure 5.1(a), then no code will be considered to be unreachable. This is not accurate as, e.g., instruction `E` is unreachable. If we apply it on Figure 5.1(b), instructions `E` and `F` are considered to be unreachable, while Figure 5.1(c) would yield `G` and `E`. However, both `F` and `G` are reachable. Therefore in this case, the result is not conservative.

We can however still maintain the formal definition of a CFG: a CFG is a directed graph $G(V, E)$ which consists of a set of vertices V, basic blocks, and a set of edges E, which indicate possible flow of control between basic blocks. A basic block is defined to be a sequence of instructions for which every instruction in a certain position dominates all those in later positions, and no other instruction executes between two instructions in the sequence.

The concept of an edge remains unchanged as well: a directed edge is drawn from basic block a to basic block b if we conservatively assume that control can flow from a to b. The CFG for our running example is shown in Figure 5.3.

In essence, this CFG is a superposition of the different CFGs observed at different times. In the middle of Figure 5.3, we can easily detect the CFG of Figure 5.1(a). The CFG of Figure 5.1(b) can also be found: just mask away instruction `D` and `H`. Finally, the CFG of Figure 5.1(c) can be found by masking instruction `C` and `D`. We will postpone the discussion of the construction of this CFG given the binary representation of the program to Section 5.1.2. For now, note that, while this CFG does represent the one possible execution (`A,B,C,G,B,H,F`), it also represents additional executions that will never occur in practice. This will be optimized in Section 5.1.3.

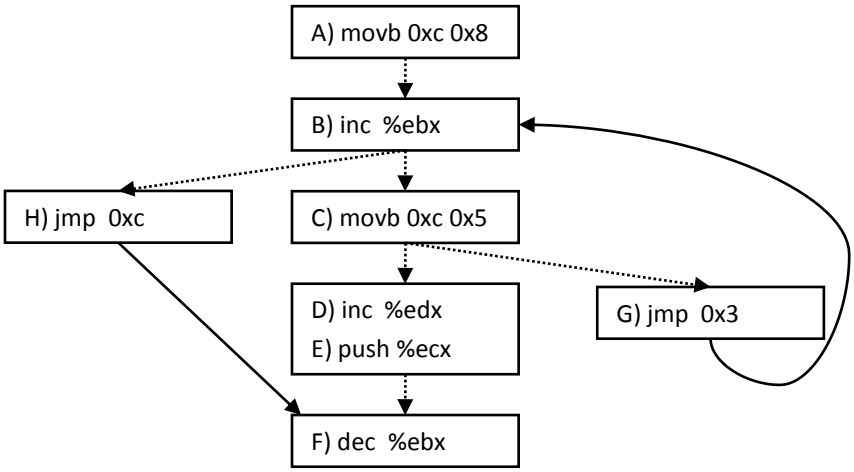


Figure 5.3: The CFG of the running example (before optimization)

Extension 1: Codebytes

The CFG in Figure 5.3 satisfies the basic property of a CFG: it represents a superset of all possible executions. As such it can readily be used to reason about a superset of all possible program executions. Unfortunately, this CFG does not yet have the same usability we have come to expect of a CFG.

One of the shortcomings is that it cannot easily be linearized. There is no way to go from this CFG to the binary representation, simply because this CFG does not contain sufficient information.

For example, there are two fall-through paths out of block `B`. Note that we follow the convention that a dotted arrow represents a fall-through path, meaning that the two connected blocks need to be placed consecutively. Clearly, in a linear representation, only one of these successors can be placed after the increment instruction. Which one should we then choose?

Finally, although this is not the case in our running example, two instructions may need to be placed in overlapping locations, while there is no indication of this constraint in the CFG.

To overcome this and other related problems, we will augment the CFG with a data structure, called codebytes. This data structure will allow us to reason about the different states of the program. Furthermore, it will indicate which instructions overlap and what the initial state of the program is.

In practice, there is one codebyte for every byte in the code segment. This codebyte represents the different states the byte can be in. By convention, the first of these states represents the initial state of that byte, i.e., the one that will end up in the binary representation of the program. For every instruction, there is a sequence of states representing its machine code. For our running

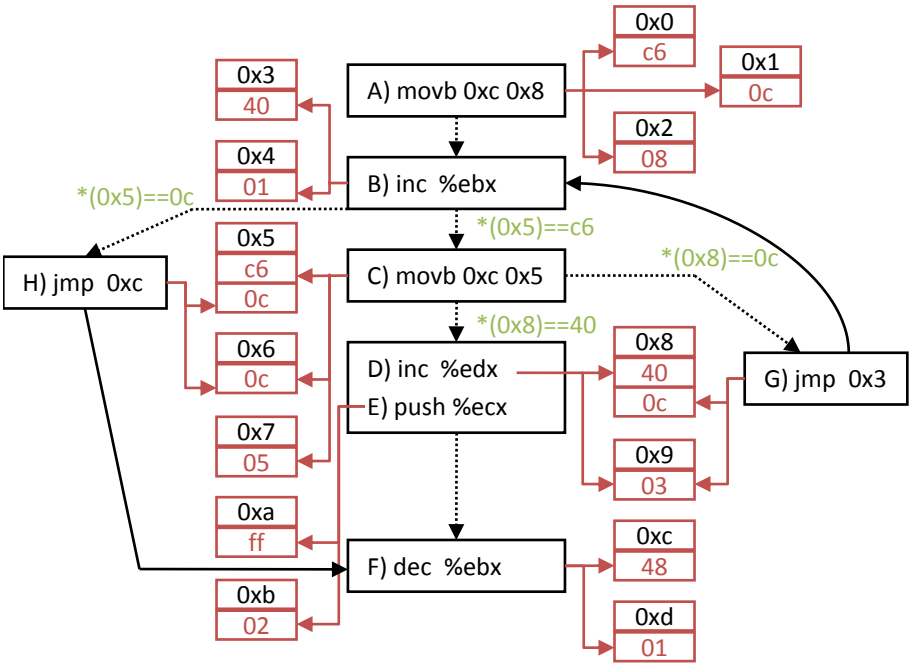


Figure 5.4: The SE-CFG of the running example (before optimization)

example, this is illustrated in Figure 5.4. We can see that instruction A and C occupy three codebytes, while the others occupy two codebytes. A codebyte consists of one or more states. For example, codebyte `0x0` has one state: $c6$ and codebyte `0x8` has two states: 40 and $0c$. We can also see that instruction `H` and `C` overlap as they have common codebytes. As the first state of codebyte `0x5` is that of instruction `C`, and the other states are identical, instruction `C` will be in the binary image of the program, while instruction `H` will not.

Codebytes are not only useful for the representation of the static code section, but also for the representation of code that could be generated in dynamically allocated memory. A region of memory can be dynamically allocated and filled with bytes representing a fragment of code which will be executed afterwards. The difference between a codebyte representing a byte in the static code section and a codebyte representing a byte that will be dynamically produced at run time is that it has no initial state because the byte will not end up in the binary representation of the program.

Extension 2: Codebyte Conditional Edges

We have repeatedly stressed the importance of having a superset of all possible executions. Actually, we are looking for the exact set of all possible executions, not a superset. In practice, it is hard, if not impossible, to find a finite representation of all possible executions and no others. The CFG is a compromise in the sense that it is capable of representing all possible executions, at the cost of representing executions that cannot occur in practice. Therefore, analyses on the CFG are conservative, but may be less accurate than optimal because they are safe for executions that can never occur.

A partial solution to this problem consists of transforming the analyses into path-sensitive variants. These analyses are an attempt to not take into account certain non-realizable paths.

Clearly, for every block with multiple outgoing paths, only one will be taken at a given point in the execution. For constant code, the chosen path may depend upon a status flag (conditional jump), a value on the stack (return), the value of a register (indirect call or jump), and so on. However, once the target of the control transfer is known, it is also known which instruction will be executed next. For self-modifying code, the target address alone does not determine the next instruction to be executed. The values of the target locations determine the instruction that will be executed as well. To take this into account, we introduce additional conditions on edges. These conditions can be found on the arrows itself in Figure 5.4. As instruction `B` is not a control transfer instruction, control will flow to the instruction at the next address: $0x5$. For constant code, this would determine which instruction is executed next: there is at most one instruction at a given address. For self-modifying code, this is not necessarily the case. Depending on the state of the program, instruction `B` can be followed by instruction `C` (*(0x5)==c6) or instruction `H` (*(0x5)==0c).

Extension 3: Consumption of Codebyte Values

The third, and final extension is designed to model the fact that when an instruction is executed, the bytes representing that instruction are read by the CPU. Therefore, in our model, an instruction uses the codebytes it occupies. This will enable us to treat code as data in data flow analyses. For example, if we want to apply liveness analysis on a codebyte, we have the traditional uses and definitions of that value: it is read or written by another instruction. For example, codebyte `0x8` is defined by instruction `A`. On top of that, a codebyte is used when it is part of an instruction, *e.g.*, codebyte `0x8` is used by instruction `D` and `G`. Note that this information can be deduced from the codebyte structure.

Wrap-up

The SE-CFG still contains a CFG and therefore, existing analyses which operate on a CFG can be readily applied to an SE-CFG. Furthermore, code can be treated exactly the same way as data: the initial values of the codebytes are written when the program is loaded, they can be read or written just as any other memory location and are also read when they are executed.

Note that in our model traditional code is just a special case of self-modifying code. The extensions can be omitted for traditional code as: (i) the code can easily be linearized since instructions do not overlap, (ii) the target locations of control transfers can only be in one state, and (iii) the result of data analyses on code are trivial as the code is constant.

Where possible, we will make the same simplifications. For example, we will only add constraints to arrows where necessary and limit them to the smallest number of states to discriminate between different successors.

5.1.2 Construction and Linearization of the SE-CFG

In this section, we discuss how an SE-CFG can be constructed from assembly code. Next, it is shown how the SE-CFG representation can be linearized.

SE-CFG Construction

Static SE-CFG construction is only possible when we can deduce sufficient information about the code. If we cannot detect the targets of indirect control transfers, we need to assume that they can go to any byte of the program. If we cannot detect information about the write instructions, we need to assume that any instruction can be at any position in the program. This would result in overly conservative assumptions, hindering analyses and transformations.

When looking at applications of information hiding, it is likely that attempts will have been made to hide this information. It is nevertheless useful to devise an algorithm for SE-CFG construction, because there are applications of self-modifying code outside the domain of information hiding which do not actively try to hide such information. Furthermore, reverse engineers often omit the requirement of proven conservativeness and revert to approximate, practically sound information. Finally, such an algorithm could be used to extend dynamically obtained information over code not covered in observed executions. For programs which have not been obfuscated deliberately, linear disassembly works well. As a result, the disassembly phase can be separated from the flow graph construction phase. However, when the code is intermixed with data in an unpredictable way, and especially when attempts have been made to thwart linear disassembly [Linn 03], it may produce wrong results. Kruegel et al. [Kruegel 04] introduce a new method to overcome most

```
00: proc main()
01: for ( addr = code.startAddr; addr ≤ code.endAddr; addr++)
02:       codebyte[addr].add(byte at address addr);
03: while (change)
04:       MarkAllAddressesAsUnvisited();
05:       Recursive(code.entryPoint);
06: proc Recursive(addr)
07: if (IsMarkedAsVisited(addr)) return;
08: MarkAsVisited(addr);
09: for each (Ins) — Ins can start at codebyte[addr]
10:       DisassembleIns(Ins);
11:       for each (v,w) — Ins can write v at codebyte w
12:          codebyte[w].add(v);
13:       for each (target) — control can flow to target after Ins
14:          Recursive(target);
```

Figure 5.5: Recursive traversal disassembly algorithm for self-modifying code

of the problems introduced by code obfuscation but the method is not useful when a program contains self-modifying code. To partially solve this problem, disassembly can be combined with the control flow information. Such an approach is recursive traversal. The extended recursive traversal algorithm which deals with self-modifying code is provided in Figure 5.5

Disassembly starts at the only instruction that will certainly be executed as represented in the static image of the program: the entry point (line 5). When multiple instructions can start at a codebyte, all possible instructions are disassembled (line 9, codebyte `0x8` in Figure 5.6(a)). When an instruction modifies the code, state(s) are added to the target codebyte(s) (line 11-12). This is illustrated in Figure 5.6(a): state $0c$ is added to codebyte `0x8`. Next, all possible successors are recursively disassembled (line 13-14). In our example, the main loop (line 3) will be executed three times, as the second instruction at codebyte `0x5` will be missed in the first run. It will however be added in the second run. In the third run, there will be no further changes. The overall result is shown in Figure 5.6(b).

The construction of the SE-CFG is straightforward once the instructions have been detected: every instruction `I` is put into a separate basic block $basicblock_I$. If control can flow from instruction `I` to codebyte c, then for every instruction `J` that can start at c, an edge $basicblock_I \rightarrow basicblock_J$ is created. Finally, basic blocks are merged into larger basic blocks where possible. The thus obtained SE-CFG for our running example is shown in Figure 5.4. Note that it still contains instructions that cannot be executed and

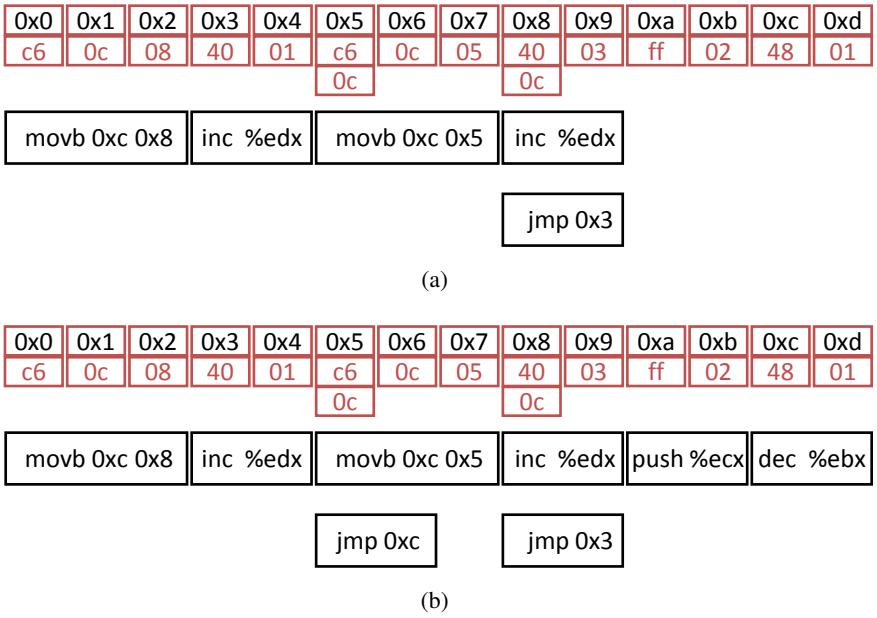


Figure 5.6: Operation of recursive traversal disassembly for self-modifying code

edges that cannot be followed. Section 5.1.3 discusses how these instructions can be pruned.

SE-CFG Linearization

Traditional CFG linearization consists of concatenating all basic blocks that need to be placed consecutively in chains. The resulting chains can then be ordered arbitrarily, resulting in a list of instructions which can be assembled to obtain the desired program.

When dealing with self-modifying code, we cannot simply concatenate all basic blocks that need to be placed consecutively and write them out. One of the reasons is that this is impossible when dealing with multiple fall-through edges. Instead, we will create chains of codebytes. Two codebytes need to be concatenated if one of the following conditions holds: (i) c and d are successive codebytes belonging to an instruction, (ii) codebyte c is the last codebyte of instruction `I` and codebyte d is the first codebyte of instruction `J` and `I` and `J` are successive instructions in a basic block, and (iii) codebyte c is the last codebyte of the last instruction in basic block A and d is the first codebyte of the first instruction in basic block B and A and B need to be placed consecutively because of a fall-through path.

The resulting chains of codebytes can be concatenated in any order into a single chain. At this point, the final layout of the program has been determined, and all relocated values can be computed. Next, the initial states of the codebytes can be written out.

For example, in Figure 5.4, codebyte `0x0`, `0x1` and `0x2` need to be concatenated because of condition (i), codebyte `0x9` and `0xa` because of condition (ii) and codebyte `0x4` and `0x5` because of condition (iii). When all conditions have been evaluated, we obtain a single chain. If we write out the first state of every codebyte in the resulting chain, we obtain the binary code listed in Section 5.1.

5.1.3 Analyses on and Transformations of the SE-CFG

In this section, we will demonstrate the usability of the SE-CFG representation by showing how it can be used for common analyses and transformations. We will illustrate how issues concerning self-modifying code can be mapped onto similar issues encountered with constant code in a number of situations.

Note that once the SE-CFG is constructed, the eventual layout of the code is irrelevant and will be determined by the serialization phase. Therefore, the addresses of codebytes are irrelevant in this phase. However, for the ease of reference, we will retain them. In practice, addresses are replaced by relocations.

Constant Propagation

The CFG of Figure 5.3 satisfies all requirements of a CFG: it is a superset of all possible executions. As this CFG is part of the SE-CFG in Figure 5.4, analyses which operate on a CFG can be reused without modifications. This includes constant propagation and liveness analysis.

Because of the extensions, it is furthermore possible to apply existing data analyses on the code as well. This can be useful when reasoning about self-modifying code. A common question that arises when dealing with self-modifying code is: “What are the possible states of the program at this program point?”. This question can be answered through traditional data analyses on the codebytes, e.g., constant propagation.

If we would perform constant propagation on codebyte `0x8` on the SE-CFG of Figure 5.4, we can see that codebyte `0x8` it is set to 40 when the program is loaded. Subsequently, it is set to $0c$ by instruction `A`. Continuing the analysis, we learn that at program point `C` it can only contain the value $0c$. Therefore, the edge from instruction `C` to instruction `D` is unrealizable, since the condition `*(0x8)==40` can never hold. The edge can therefore be removed.

Unreachable Code Elimination

Traditionally, unreachable code elimination operates by marking every basic block that is reachable from the entry node of the program. For self-modifying code, the approach is similar. For our running example, this would result in the elimination of basic blocks `D` and `E`. Note that the edge between `C` and `D` is assumed to have been removed by constant propagation.

Similarly, we can remove unreachable codebytes. A codebyte can be removed if it is not part of any reachable basic block and if it is not read by any instruction. This allows us to remove codebyte `0xa` and `0xb`. While we have removed the `inc %edx`-instruction, its codebytes could not be removed, as they are connected through another instruction. Note that we now have a conservative and accurate unreachable code elimination.

Liveness Analysis

Another commonly asked question with self-modifying code is as follows: "Can I overwrite a fragment of code?". Again, this is completely identical to the question whether you can overwrite a fragment of data. You can overwrite a fragment of the program if you can guarantee that the value will not be read later on by the program before it is overwritten. In our model, for self-modifying code, a value is read when (i) it is read by an instruction (standard), (ii) the flow of control can be determined by this value (extension 2), and (iii) the CPU will interpret it as (part of) an instruction (extension 3).

We could, for example, perform liveness analysis on codebyte `0x8`. This shows us that the value 40, which is written when the program is loaded, is a dead value: it is never read before it is written by instruction `A`. As a result, it can be removed and we could write the second state $0c$ immediately when the program is loaded. In our representation, this means making it the first state of codebyte `0x8`.

Subsequently, an analysis could point out that instruction `A` has now become an idempotent instruction: it writes a value that was already there. As a result, this instruction can be removed. We have now obtained the SE-CFG of Figure 5.7.

Loop Unrolling

Subsequently, we could peel off one iteration of the loop to see if this would lead to additional optimizations. This results in the SE-CFG in Figure 5.8. Note that we had to duplicate the write operation `C`, as we should now write to both the original and the copy of the codebyte in order to be semantically equivalent.

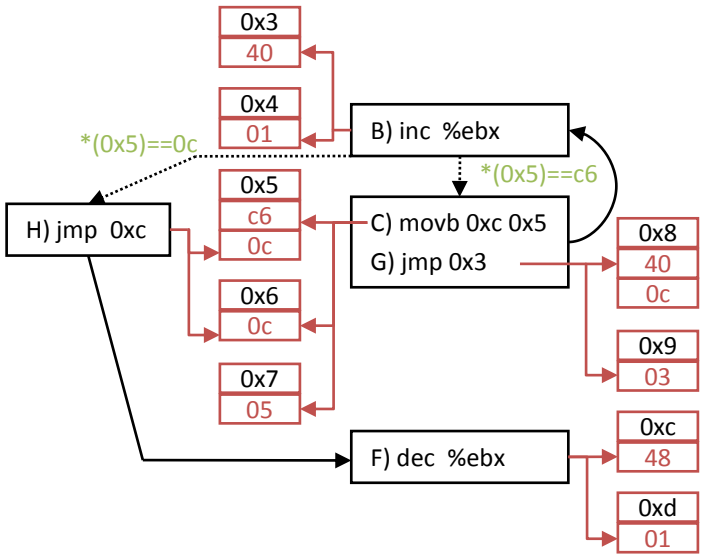


Figure 5.7: The SE-CFG after partial optimization, before unrolling

Finishing Up

Similarly to the previously discussed constant propagation, we can now find out that the paths `B'` → `H'` and `B` → `C` are unrealizable. As a result, we no longer have a loop. Instruction `C`, `C2`, `G` and `H'` are unreachable. Applying the same optimization as in Section 5.1.3, we can remove the first state of codebyte `0x5` and instruction `C'`. The value written by `C2'` is never used and thus `C2'` can be removed. Through jump forwarding, we can remove instruction `H`. Finally, given that the decrement instruction performs exactly the opposite of the increment instruction, we now see that the code can be replaced by a single instruction: `inc %ebx`.

5.1.4 Folding through Self-modifying Code

Based upon our model for self-modifying code, the SE-CFG, we have implemented another form of folding. This choice is motivated by the same arguments as given in Section 4.6.1. Again, the goal of this type of folding is not necessarily to shrink the program, but to make it more tamper resistant. As a result, in some cases, we will also perform folding if the cost (in terms of size, speed and power) is higher than the gain.

The transformation operates as follows. In the first phase, we split the code up in what we call *code snippets*. These code snippets are constructed as follows: if a basic block is not followed by a fall-through edge, the basic block

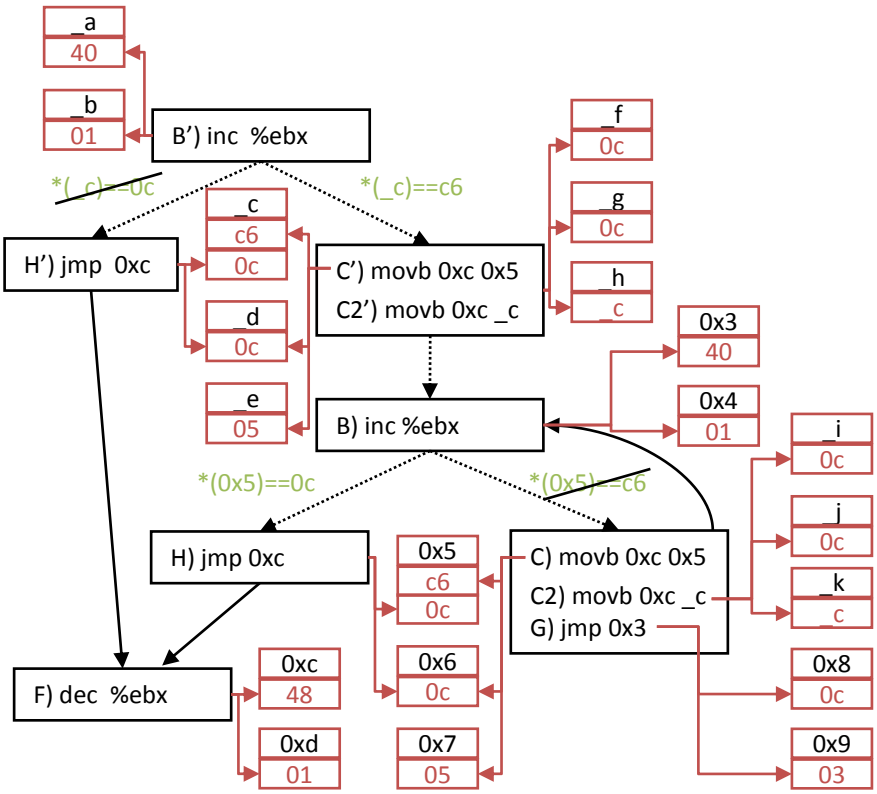


Figure 5.8: The SE-CFG after unrolling

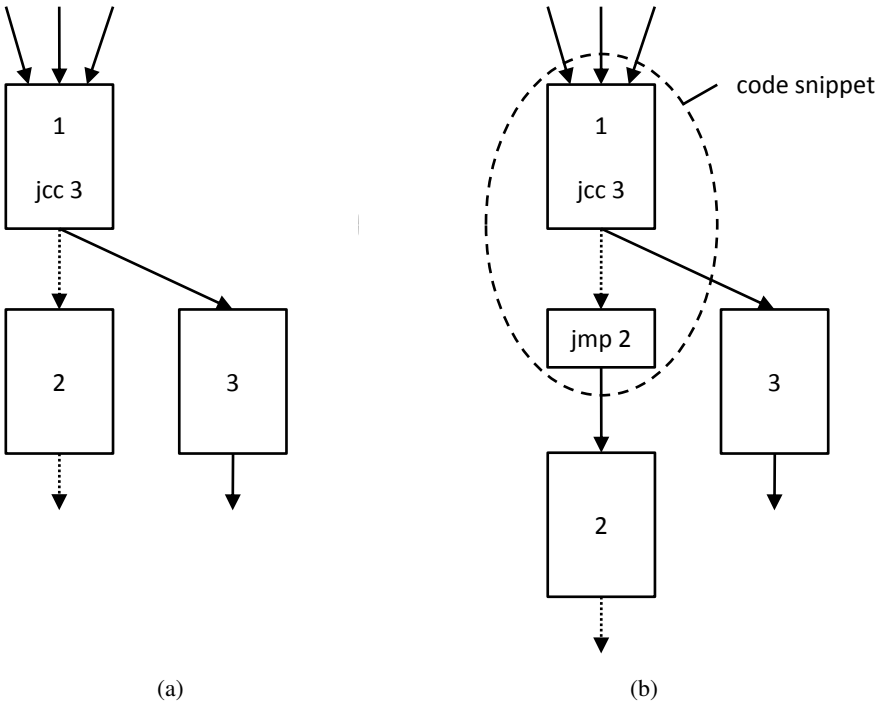


Figure 5.9: Code snippet generation

itself makes up a code snippet. If basic block a was followed by a fall-through edge e to basic block b, a new basic block c is created with a single instruction: a jump to b. The target of e is then set to c. The combination of a and c is then called a code snippet. This is illustrated in Figure 5.9.

A code snippet is thus a small fragment of code that can be placed independently of the other code. It consists of at most two consecutive basic blocks. If there is a second basic block, this second basic block consists of a single jump instruction. The advantage of code snippets is that they can be transformed and placed independently. The downside is that their construction introduces a large number of jump instructions. This overhead is partially eliminated by performing jump forwarding and basic block merging at the end of the transformation.

Next, we perform what we call code snippet coalescing. Wherever this is possible with at most one modifier, we let two code snippets overlap. Both code snippets are then replaced by at most one modifier and a jump instruction to the coalesced code snippet. On the 80x86, this means that code snippets are merged if they differ in at most 4 consecutive bytes.

As an example, consider the two code snippets in Figure 5.10(a). While these two code fragments seem to have little in common, their binary representation differs in only one byte. Therefore, they are eligible for code snippet coalescing. The result is shown in Figure 5.10(b). The codebytes of the modifier and jump instructions are not shown to save space. In this example, subsequent branch forwarding will eliminate one of the jumps. (Note that this example uses the actual 80x86 instruction set.)

Intuitively, this makes the program harder to understand for a number of reasons. Firstly, as overlapping code snippets are used within multiple contexts, the number of interpretations of that code snippet increases. It also becomes more difficult to distinguish functions as their boundaries have been blurred. And most importantly, the common difficulties encountered for self-modifying code have been introduced: the code is not constant and therefore, the static image does not contain all the instructions that will be executed. Furthermore, multiple instructions will be executed at the same address, so there is no longer a one to one mapping between addresses and instructions.

Folding with a One-byte Modifier

For each pair of code snippets that differs in exactly one byte, we can choose whether or not to merge them using the above described transformation. Note that this can be done repeatedly if a next code snippet differs in the same byte as an earlier merged pair of code snippets.

Folding with a Four-byte Modifier

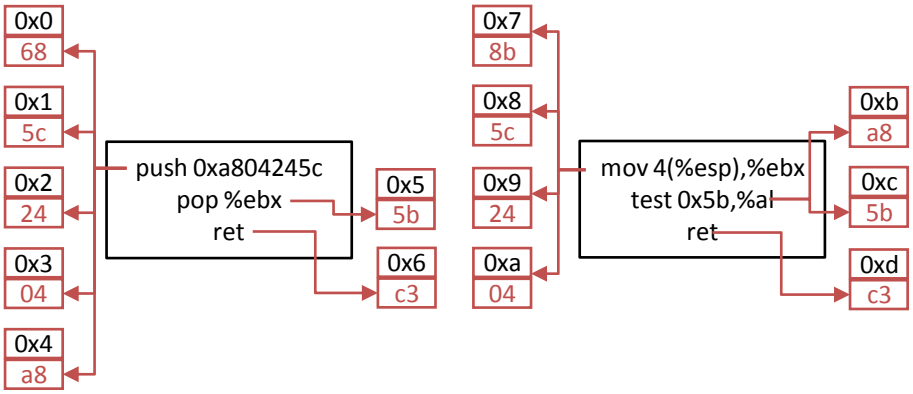
For each pair of code snippets that differs only in the offset of the final control transfer instruction (e.g., `jmp` and `call`), we can choose whether or not to merge them using the above described transformation. Note that this can be done repeatedly if a next code snippet differs in the same way as an earlier merged pair of code snippets.

5.1.5 Evaluation

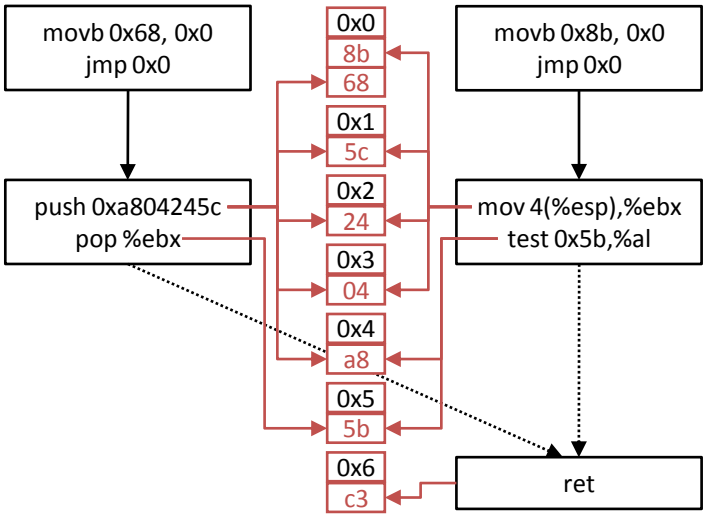
The number of opportunities for the folding transformations based on self-modifying code are shown in Table 5.1.

In Figure 5.11, we have plotted the cost of these transformations in terms of code size and execution time. On the x-axis, we have assigned different values to p, the probability with which a folding transformation is applied. We have chosen to use the same p for the two transformations at the same time.

Despite the folding nature of these transformations, we can still observe a marginal increase in code size (0.7% for $p = 1$). The reason is that the



(a) Snippets to coalesce



(b) Coalesced snippets

Figure 5.10: Example of coalescing code snippets

	No restrictions		With restrictions	
benchmark	1-byte	4-byte	1-byte	4-byte
400.perlbench	1029	13384	422	1530
401.bzip2	93	3784	20	229
403.gcc	1917	31671	1098	5553
429.mcf	273	3386	32	137
433.milc	507	3944	54	243
445.gobmk	1603	8009	564	1923
456.hmmer	417	4229	46	252
458.sjeng	338	4039	44	356
462.libquantum	325	3512	30	102
464.h264ref	368	6100	121	585
470.lbm	310	3393	21	77
482.sphinx	415	3181	85	467

Table 5.1: Number of candidates for self-modifying code per benchmark

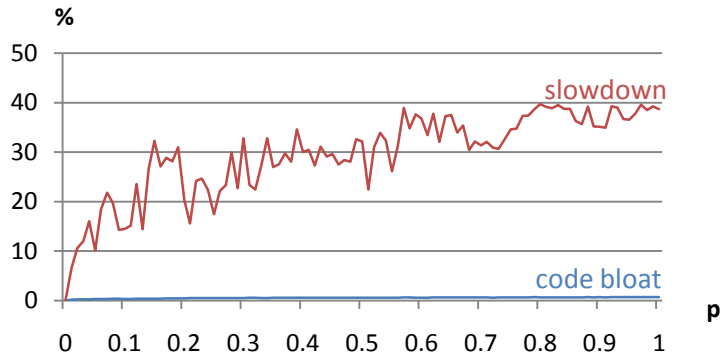


Figure 5.11: Code bloat and slowdown for self-modifying code

reduction in code size is undone by the large size of the modifying instructions. A one-byte modifier takes seven bytes, while a four-byte modifier takes ten bytes. Note that we apply the transformation even if there is no net gain, as our goal is not compaction.

The cost in terms of execution time is considerable. However, it is still small compared to the slowdown we expect an attacker to experience, e.g., if techniques such as dynamic instrumentation are being used.

Slowdown of Information Collection

Most tools for program analysis assume that code is constant and have little or no provisions for self-modifying code.

When using a dynamic instrumentor, we need to monitor every write instruction to see if it doesn't modify any already instrumented code in order to correctly deal with self-modifying code. Furthermore, if already instrumented code is modified, we need to invalidate and re-instrument the related code fragments. Finally, when executing instrumented code, we will want to know to what version the executed code corresponds to, to map it correctly to the instructions and basic blocks in the CFG. This requires some additional bookkeeping as well.

DIOTA already contained support for this type of code. Yet, little measures were taken to minimize the execution overhead. As a result, we were unable to collect the information in reasonable time. Monitoring every write instruction causes a slowdown of a factor 2. Invalidation and re-instrumentation results in an additional slowdown of a factor of more than 200 for the discussed transformations based on self-modifying code.

Clearly, this does not prove that the information cannot be collected faster. Another, more advanced analysis, or even attempts to rewrite the code in a non-self-modifying version may accelerate the process. However, given the general consensus that self-modifying code is hard to deal with, we assume that this is one of the ways to significantly increase the attacker's workload for a relatively moderate cost.

5.2 Virtualization

The last diversifying and anti-tampering transformation we discuss is also the most radical transformation: virtualization. The idea is to rewrite the entire program for a custom virtual machine and ship it along with an implementation of that virtual machine. This is illustrated in Figure 5.12.

Having the freedom to design our own ISA leaves us with many choices. We would like to use this freedom of choice to create a set of different versions of the program with the following properties: (i) each version in the set has a reasonable level of defense against tampering and (ii) it is hard to retarget an existing attack against one version to work against another version.

A number of design principles to achieve the first goal are discussed in Section 5.2.1. This involves a trade-off: the many choices, discussed in Section 5.2.2, result in a large design space for ISAs. We can consider this entire space to allow for more diversity. Alternatively, we can focus on subspaces which we believe to lead to more tamper resistant programs than other parts.

We will evaluate the cost of the applied transformations in Section 5.2.3.

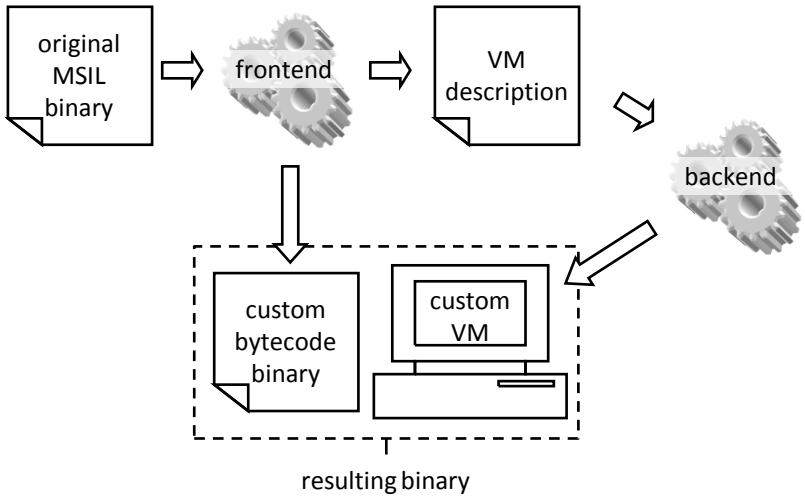


Figure 5.12: High-level overview of virtualization

5.2.1 Design Principles for an ISA Targeted at Software Protection

The design principles to steer towards tamper resistant properties are: (i) prevent static analysis of the program; (ii) prevent dynamic analysis of the program; (iii) prevent local modifications and (iv) prevent global modifications.

The first two are closely related to the problem of obfuscation, while the latter two are more tamper resistance oriented. However, intelligent tampering requires at least some degree of program understanding, which is typically gained from observing the static image of the program, observing the running program or a combination and/or repetition of the two previous techniques.

The Trend Towards RISC Architectures

It seems as if these design principles conflict with the trend in general purpose ISA design. The concept of the Complex Instruction Set Computer (CISC) has lost ground to Reduced Instruction Set Computer (RISC). Because of the complexity of CISC instruction sets, often accompanied by fewer restrictions on the program, CISC programs are usually more complex to analyze than RISC programs. The complexity of CISC architectures can complicate analysis and allows for a number of tricks that cannot be used as easily on RISC architectures. For example, Linn et al. [Linn 03] have exploited variable instruction lengths and intermixing of code and data to try to confuse the disassembler. Self-modifying code (see Section 5.1) is facilitated on the IA-32 because ex-

plicit cache flushes are not required to communicate the modifications to the Central Processing Unit (CPU).

Other research, such as control flow flattening, does not rely on architecture or CISC-specific features, and can thus be applied to RISC architectures as well.

The Trend Towards Virtualized Execution Environments

More recently, the advent of Java bytecode and managed Microsoft Intermediate Language (MSIL) has promoted ISAs that are even more easily analyzed. This is due to a number of reasons. First, these programs are typically not executed directly on hardware, but need to be emulated or translated into native code before execution. To enable this, boundaries between code and data need to be known and there can be no confusion between constant data and relocatable addresses. This, of course, comes with the advantage of portability. Besides portability, design principles include support for typed memory management and verifiability. To assure verifiability, pointer arithmetic is not allowed, control flow is restricted, etc. To enable typed memory management, a lot of information needs to be communicated to the executing environment about the types of objects.

All of these design principles have led to programs that are easy to analyze by the executing environment, but are equally easy to analyze by an attacker. This has led to the creation of decompilers for both Java (e.g., DejaVu and Mocha) and managed MSIL programs (e.g., Reflector).

As a result of this vulnerability, a vast body of valuable research can be found in the protection of Java bytecode [Badger 01, Collberg 03]. Note that most of the developed techniques are theoretically applicable to CISC and RISC programs as well, while in practice their application is complicated by the absence of rich information.

One can clearly observe a trend where the design principles of ISAs are increasingly in conflict with the design principles that would facilitate software protection. Surprisingly, one way to counter this trend is to add an additional layer of virtualization. We will emulate our own ISA. This idea has been mentioned in the academic literature as *table interpretation* [Collberg 98a].

Reusing Experience

Experience and intuition tell us that the average IA-32 program is far more complex to understand and manipulate than the average managed program. We believe that three key factors are in play: (i) variable instruction length;

(ii) no clear distinction between code and data and (iii) no clear distinction between constant data and relocatable addresses.

Since instructions (opcode + operands) can have variable length (1-15 bytes), instructions need only be byte-aligned, and can be mixed with padding bytes or data on the IA-32, disassemblers can easily get out of synchronization. This has been studied in detail by Linn et al. [Linn 03].

As there is no explicit separation between code and data, both can be read and written transparently and used interchangeably. This enables self-modifying code, which has been discussed in Section 5.1.

The feature that the binary representation of the code can easily be read has been used to enable self-checking mechanisms [Chang 02, Horne 02]. The absence of restrictions on control flow has enabled techniques such as control flow flattening [Chow 01, Wang 01] and instruction overlapping [Cohen 93].

While we know of no explicit publication which specifically exploits this feature, the fact that addresses can be computed, and that they cannot easily be distinguished from regular data, complicates the tampering with programs: an attacker can only make local modifications, as he does not have sufficient information to relocate the entire program.

These observations are an important inspiration when looking for ways to generate a hard to analyze ISA. We will highlight subspaces which lead to increased tamper resistance when discussing the available choices.

5.2.2 Available Choices

We have identified a number of different choices based on the operation of a Virtual Machine (VM). A high-level overview of the execution engine is provided in Figure 5.13. The different components are:

1. instruction semantics;
2. opcode encoding;
3. operand encoding;
4. fetch cycle;
5. program counter and program representation.

In order to keep the diversity manageable, we have fixed the interface of the different components of the ISA. As a result, we can diversify their internal implementation regardless of the transformations applied on the other components. This allows for a modular design and independent development.

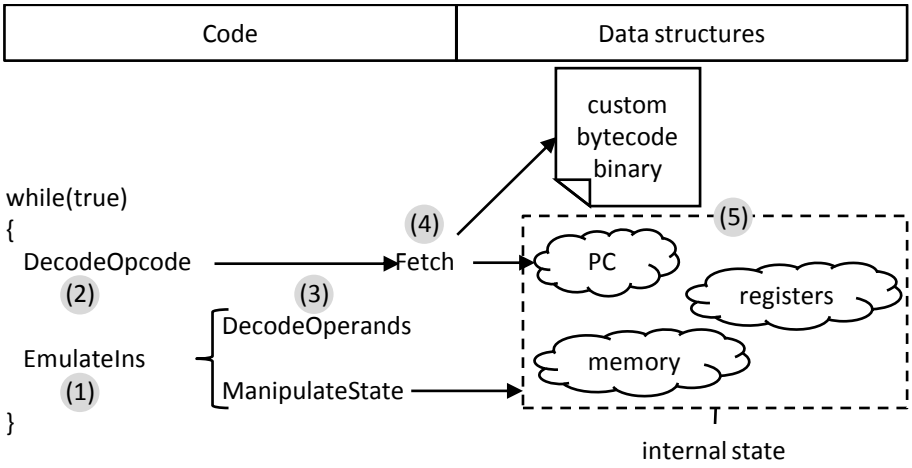


Figure 5.13: The execution model and the interfaces of the virtual machine

The arrows in the Figure 5.13 indicate interface dependencies. For example, DecodeOpcode expects to be able to fetch a number of bits. The main internal data structures of the VM are shown as well.

The above components are sufficient to generate a program in the custom bytecode language, i.e., these determine the ISA. When desired, the virtual machine itself can be diversified as well. Additionally, we can also diversify the code of the original program before the layer of virtualization is added.

Instruction Semantics

To allow for the diversification of instruction semantics, we use the concept of micro-operations. An instruction in the custom bytecode language can be any sequence of a predetermined set of micro-operations. The set of micro-operations can, for example, include the instructions of the original architecture and a number of additional instructions to: (i) communicate meta-information required for proper execution and (ii) enable additional features such as changing semantics (see Section 5.2.2). This can be compared to the concept of micro-operations (μops) in the P6 micro-architecture [Patterson 90]. Each IA-32 instruction is translated into a series of μops which are then executed by the pipeline. This could also be compared to the super-operators by Proebsting [Proebsting 95]. Super-operators are virtual machine operations automatically synthesized from smaller operations to avoid costly per-operation overheads and to reduce program size.

We have provided stubs to emulate each of the micro-operations and these can simply be concatenated to emulate more expressive instructions in our custom bytecode language.

Tamper-resistance Not knowing the semantics of an instruction will complicate program understanding. We can however go one step further and choose our instruction semantics to adhere to some design principles for a tamper resistant ISA.

Conditional execution We can use conditional execution to further promote merging slightly differing fragments of code. In the presence of conditional execution, instructions can be predicated by predicate registers. If the predicate register is set to false, the instruction is interpreted as a no-op, otherwise it is emulated. The idea is to set these registers on or off along different execution paths to be able to outline slightly different fragments of code.

Limited instruction set The VM is tailored to a specific program. Therefore, we can make sure that the VM can only emulate operations that are required by that program. We can further limit the instruction set. A common way for an attacker to remove undesired functionality (e.g., a license check) is to overwrite that functionality with no-ops. There is little reason to include a no-op instruction in our custom ISA and not having this instruction will complicate padding out unwanted code.

Statistics furthermore show that, for example, of the integer literals from some 600 Java programs, 1.4 million lines in all, 80% are between 0-99, 95% are between 0 and 999, and 92% are powers of two or powers of two plus or minus 1 [Cousot 03]. This allows us to limit the number of operands that can be represented, again limiting the freedom of the attacker.

Another example can be found with conditional branches. Usually, there are two versions for each condition: branch if condition is set and branch if condition is not set. Since this is redundant, we could rewrite the code so that only one version is used and not include its counterpart in the ISA. This may be useful, for example, when a license check branches conditionally depending on the validity of the serial number: it will prevent the attacker from simply flipping the branch condition.

Opcode Encoding

Once instruction semantics has been determined, we need to determine an opcode encoding for those instructions. The size of all opcodes for traditional architectures is usually constant or slightly variable. For example, MSIL opcodes are typically one byte, with an escape value (0xfe) to enable two-byte

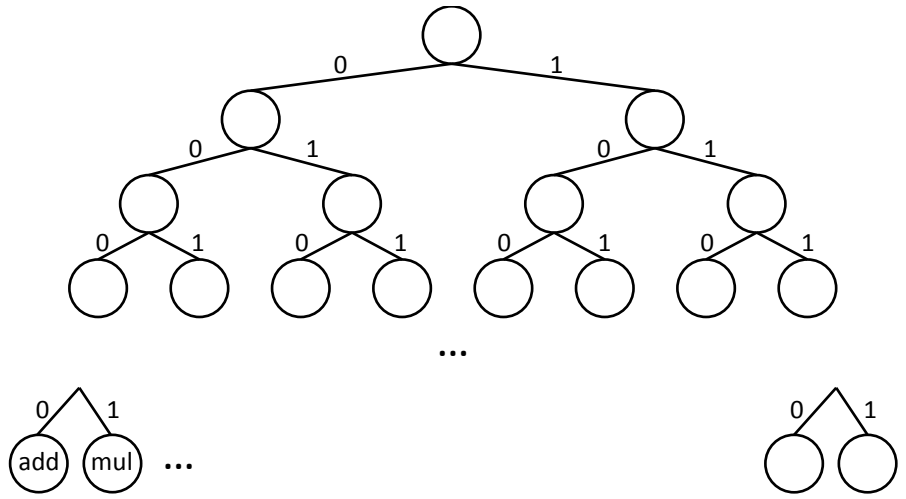


Figure 5.14: Prefix code decoding with a binary tree

opcodes for less frequent instructions. The limited variability facilitates fast lookup through table interpretation. But, more generally, any prefix code (no code word is a prefix of any other code word) allows for unambiguous interpretation.

In its most general form, decoding opcodes to semantics can be done through a binary-tree traversal. Decoding starts in the root node; when a `'0'` bit is read, we move to the left child node; when a `'1'` bit is read, we move to the right child node. When a leaf node is reached, we have successfully decoded an opcode. This is illustrated in Figure 5.14. The leaf node contains a reference to the `case`-statement emulating the semantics of the instruction.

If we allow arbitrary opcode sizes, without illegal opcodes, the number of possible encodings for n instructions is:

$$\frac{\binom{2(n-1)}{n-1}}{n} n! \ .$$

The fraction represents the number of planar binary trees with n leaves (Catalan number), while the factorial represents the assignment of opcodes to leaves.

If we choose fixed opcode sizes with the shortest possible encoding, i.e. $\lceil \log_2(n) \rceil$ bit, we might introduce illegal opcodes. In this case, the number of possible encodings is:

$$\binom{2^{\lceil \log_2(n) \rceil}}{n} \ .$$

Many more possibilities would arise if we allowed illegal opcodes for other reasons than minimal fixed opcode sizes. However, this increases the size of a program written in the custom ISA without any clear advantages. Therefore we do not consider this option.

We currently support the following modes: (i) fixed-length opcodes with table lookup; (ii) multi-level table encoding to enable slightly variable instruction sizes (escape codes are used for longer opcodes) and (iii) arbitrary-length opcodes with binary-tree traversal for decoding.

Tamper-resistance Again, not knowing the mapping from bit sequences to semantics introduces a learning curve for the attacker, as opposed to having that information in a manual. Again, there are a number of additional tricks to choose this mapping in such a way that it allows for tamper resistance properties.

Variable instruction sizes We already know that variable instruction sizes introduce complexity in disassembling CISC programs. When designing our own ISA, we can introduce even more variance in the length of opcodes.

Variable instruction sizes can also be used to make local modifications more complicated. It is easy to see how a larger instruction cannot simply replace a smaller instruction, because it would overwrite the next instruction. We can also make sure that smaller non-control-transfer instructions cannot replace larger instructions. This can be done by making sure that they cannot be padded out to let control flow to the next instruction.

For example, if we have 64 instructions, we could assign each of them a unique size between 64 and 127 bits. Clearly, larger instructions do not fit into the space of smaller instructions. Smaller instructions do fit in the space of larger instructions, but when control falls through to the next bit, a problem arises: there is no instruction available to pad out the remaining bits with no-ops to make sure that control flows to the next instruction. Under this scheme it is useful to make control-transfer instructions the longest, to keep an attacker from escaping to another location where he can do what he wants.

Unary encoding To entangle the program further, we could try to maximize physical overlap. We want to be able to jump into the middle of another instruction, and start decoding another instruction. We could facilitate this by choosing a good encoding. For example, we could use unary encoding to encode the opcodes $(0, 01, 001, ..., 0^{63}1)$; there is a good chance that we find another instruction when we jump one bit after the beginning of an instruction. This is illustrated in Figure 5.15. Four instructions have been assigned an opcode using unary encoding. We can see that if decoding is started at the second

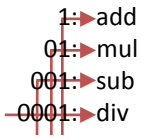


Figure 5.15: Unary encoding to promote physical overlap

bit of the 'divide' instruction, the 'subtract' instruction is revealed. Likewise, looking at the last bit of the 'divide', 'subtract' and 'multiply' instruction reveals the 'add' instruction.

Non-local semantics Having a unique bytecode language for every distributed copy is clearly a major barrier for attackers. There is no documentation available on: (i) the mapping from bit patterns to instructions; (ii) the semantics of instructions; (iii) the mapping from bit patterns to operands; (iv) the representation of data structures; etc.

However, this information can eventually be obtained through static or dynamic inspection. We can further complicate this process by making sure that a bit pattern has different meaning along different execution paths.

A program is just a sequence of `'1'`s and `'0'`s, which is given meaning by the processor. The meaning between bit patterns and interpretation is typically fixed by the ISA. On traditional architectures, if the opcode of a certain instruction is represented by a given bit pattern, this pattern is constant for every program, everywhere it occurs. We want to make this variable.

A bit pattern should only be assigned meaning depending on previously executed code. The first observation we need to make if we want the interpretation to depend on previously executed code is that, depending on the (fully specified) input, we can get to a program point along different execution paths. However, we still want to have control over the interpretation of bits at a given program point. To accommodate this variability, we have chosen to make interpretation changes explicit in the ISA, rather than implicit as a side effect of some other event.

Another consideration that we need to make is that it should not be overly complex to get the executing environment in a specific interpretation state, so that if we can get to a program point from different execution paths in different interpretation states, we can relatively easily migrate to a single target interpretation state no matter what those different interpretation states are.

The approach we have taken is the result of the following observation: if we look back at Figure 5.14, it is easy to see that changing interpretation is nothing more than rearranging the decoding tree.

Taking into account the previous observations, we can only allow a limited form of diversification. To this end, we have chosen a level at which subtrees can be moved around. This choice is a trade-off between how many different interpretations are possible and how easy it is to go to a fixed interpretation from a set of possibly different interpretation states. We have chosen the third level. Assuming that the shortest opcode is 3 bit, this allows for $8!$ interpretation states, while any interpretation state is reachable in at most 8 micro-operations.

The micro-operations we have added to the set of micro-operations to enable this are:

- `Swap(UInt3 position1, UInt3 position2)`, which exchanges the nodes at position`position1` and `position2` and
- `Set(UInt3 label, UInt3 position)`, which exchanges the node with label `label`(wherever it may be) and the node at position `position`.

In the case of table interpretation, this is implemented as a two-level table interpretation. The first level simply refers to other tables which can be swapped.

Operand Encoding

As our micro-operations largely correspond to the instructions of the original architecture, the operand types correspond largely to the operand types in the original architecture. Micro-operation emulation stubs that use operands use function calls to ensure that opcode encoding can be diversified orthogonally to what we have previously discussed. These callbacks furthermore pass an argument `insNr` identifying the custom VM instruction from which it was called. This allows us to encode operands differently for different custom VM instructions. Note that due to the concatenation of stubs, an arbitrary number of operands can follow the opcode.

Similar observations on diversifying the opcode encoding can be made as for instruction encoding.

Fetch Cycle

Diversifying the fetch cycle is really an artificial form of diversification. In its most simple form, the fetch cycle simply gets a number of bits from the custom bytecode program, depending on the current Program Counter (PC). However, we will allow a number of filters to be inserted into this phase to

allow for improved tamper resistance. Basically, they will transform the actual bits in the program to the bits that will be interpreted by the VM.

These filters will typically combine the requested bits with other information. For example, the actual requested bits may be combined with other parts of the program. This way, the program becomes more inter-dependent as changing one part of the program may impact other parts as well. Other applications include combining it with a random value derived from a secret key, or combining it with the program counter to complicate pattern matching techniques.

Program Representation and Program Pointer

We are very familiar with the traditional representation of the code as a linear sequence of bytes. The program counter then simply points to the next byte to execute, and control transfers typically specify the byte to continue execution at as a relative offset or an absolute address. This could be seen as representing the code as an array of bytes.

However, it is worth noting that an array is not the only data structure that can be used to represent code. In fact, almost any data structure will do. We could represent the code as a hash table, as a linked list, as a tree structure, etc.

So far, we have implemented representing the code as a linear sequence and as a splay tree [Sleator 85]. Related research includes keeping data, as opposed to code, in a splay tree [Varadarajan 06]. Splay trees have a number of advantages: they are self-balancing, which will allow for automatic relocation of code. Furthermore, they are nearly optimal in terms of amortized cost for arbitrary sequences. Finally, recently accessed nodes tend to be near the root of the tree, which will allow us to partially leverage temporal locality.

Because of the self-balancing property, a fragment of code could be in many different locations in memory, depending on the execution path that led to a certain code fragment. Code fragments can be moved around, as long as there is a way to refer to them for control flow transfers, and we can retrieve them when control is transferred to them. We will use the keys of the nodes in the splay tree to make this possible: control transfers specify the key of the node to which control needs to be transferred.

As such, it is required that targets of control flow be nodes. We cannot jump into the middle of the code contained within a node. In practice this means that we start a new node for each basic block. We deal with fall-through paths by making all control flow explicit. All control flow targets are specified as the keys of the node containing the target code.

This is illustrated in Figure 5.16 for the factorial function. When, for example, the function is called for the first time, the node with key ***1*** will be referenced and percolated to the root, as shown in part (2).

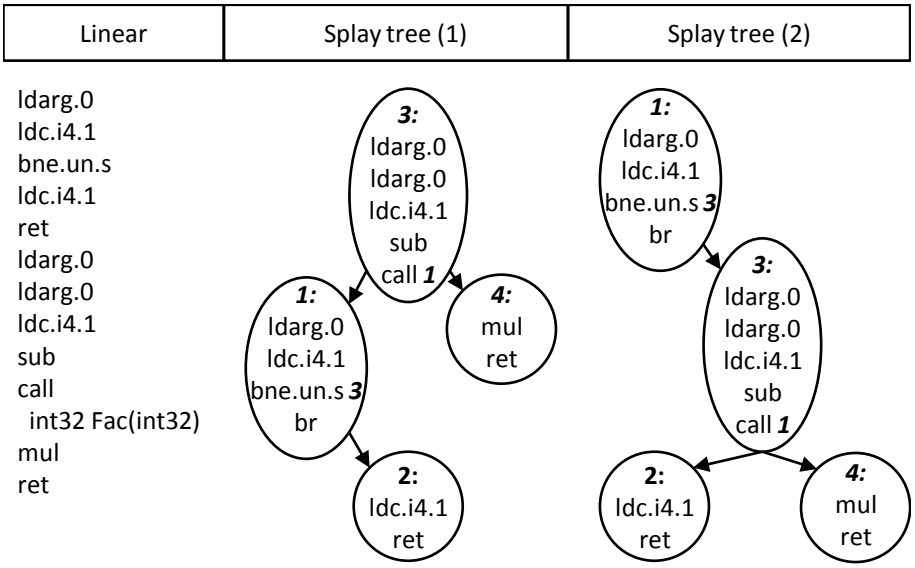


Figure 5.16: Linear versus splay tree representation for the factorial function.

benchmark:	Arith.	Cast.	Create.	Loop.	FFT	MatMult
slowdown:	50.38	284.02	1557.39	253.8	1823.39	3516.39

Table 5.2: Slowdown (factor) incurred by virtualization for C# versions of the Java Grande benchmark suite

We also want to note that if this technique is implemented naively only pointers will be moved around, and the actual code will remain at the same place on the heap. To overcome this, we can explicitly exchange the actual contents (of primitive types) of the nodes, or alternatively, we can allocate a new code buffer and copy the code buffer there, possibly with re-encryption with different garbage padding.

VM Generation

Once the choices for the above specified forms of diversification have been made, the backend will combine code fragments from various locations along with some auto-generated code to assemble a custom virtual machine, which is then shipped with the rewritten program.

5.2.3 Evaluation

The slowdown for some C# versions of benchmarks of the Java Grande Benchmark suite is shown in Table 5.2. The overhead of the techniques is considerable, ranging between a factor 50 and 3500. There are several reasons for this overhead.

Firstly, this slowdown is a worst-case slowdown. Every function in the program has been transformed. The tool has been configured to use binary tree decoding as opposed to more optimal table interpretation to leave the instruction length completely randomizable. This results in a significant overhead for every decoded bit. Furthermore, the code is represented as a splay tree, which is clearly less optimal than an array representation.

Secondly, this is a recently developed proof-of-concept evaluation framework for research in software protection. As such, it has not been optimized. Thirdly, there is an inherent overhead involved with adding an extra layer of virtualization.

There are applications in DRM and license systems where this kind of slowdown could be acceptable. These applications typically boil down to some computations followed by one or more Boolean checks. Computations that are too time-consuming (e.g., asymmetric cryptography with large keys) would need to be omitted from virtualization. Both run-time profiling and programmer input could be used to determine which parts are practically virtualizable. The significant slowdown does show us that this level of transformation will typically not be acceptable for performance-critical parts of the program.

6

Conclusion and Future Work

Even though the value of software diversity in the malicious code model was documented by Cohen as early as 1993 [Cohen 93], it appears to only have gained significant attention in the security community very recently. The end user can benefit from diversity through ASLR in Windows Vista, Mac OS X 10.5 and Linux (via PaX). The interest in the topic from academia is illustrated by publications at major venues such as the ACM Conference on Computer and Communications Security ([O'Donnell 04, Shacham 04]) and Usenix Security ([Sovarel 05]).

Diversity for software protection has initially been focused on protecting benign environments against malicious code. This is similar to many other areas in computer science. For example, computer security has originally been about "keeping the bad guys out". More recently, software protection techniques, such as obfuscation and tamper resistance, have been studied as a defense against malicious end users.

Likewise, cryptography has initially assumed that the execution environment could be trusted. The goal of the more recent direction of white-box cryptography is to be able to deal with hostile execution environments as well.

Similarly, software diversity as a defense against malicious hosts has received far less attention that its counterpart in the malicious code model. The work covered in this dissertation is among the first publicly available literature in this domain and we expect to see many interesting developments in the future.

6.1 Summary

We started with an extensive discussion of the potential benefits of diversity. From a simplified multi-phased economical model, we derived that diversity can limit the number of illegitimate users in the presence of a successful attack against one or a limited number of copies. Furthermore, diversity makes it possible to discriminate between legitimate and illegitimate copies and enables us to decrease the value of an illegitimate copy over time by restricting access to updates, additional content and features to legitimate copies.

In an alternative setting, where diversity is introduced at run time, we have used a model of the tamperer's behavior, the locate-alter-test cycle, to discuss how diversity can be used to delay the locate phase by making it harder to zoom in on the origin of undesired behavior. Diversity can furthermore delay the test phase by making the software behave differently for different hardware, different input types, different days, etc.

As a result of the novelty of this research direction, there is no consensus on how to evaluate this type of techniques. Ideally, we would be able to prove some property such as "an attacker can learn no more from this version about another version than its I/O behavior" or "the combined time of attacking each copy in isolation is smaller than the time required to set up a generic attack". In practice, this goal is not yet within reach. As an alternative way to quantify techniques that do intuitively increase the diversity between versions, we have defined an approximative metric.

This metric describes how successful the diversity is at fooling a first automated step in a practical collusion attack: a matching system. The goal of a matching system is to identify related pairs of code fragments from the two versions. The matching system can make two types of errors: identifying code fragments as related that are not related (false positives) and failing to identify code fragments that are related (false negatives). The false positive and false negative rate are then used as an indication of the quality of the diversity.

Experimental evaluation shows that we are able to fool the matching system to a large extent by combining a number of smaller transformations from different domains. The false positive rate was increased to 0.58, while the false negative rate was increased to 0.76.

Through the use of self-modifying code, we are able to significantly slow down the matching system. The time required to collect the information used by the matching system is increased by a factor of over 400 if it has to deal with fine-grained self-modifying code conservatively. Part of this slowdown can be explained because the tool has been optimized for the average case: constant code. Yet, a significant slowdown is likely to remain after optimization for self-modifying code.

Self-modifying code is considered bad practice by many and some operating systems go through great lengths to rule it out. The main goal is to defend against some buffer overflow attacks. However, as self-modifying code has been, and still is, used for optimization, run-time code generation and software protection, it can be used by taking the necessary precautions. Furthermore, on some architectures, e.g., the Pentium architecture, the usage of self-modifying code is completely transparent, i.e., no explicit cache flushes are required.

Finally, through virtualization, we are able to undermine the basic operation of the matching system: matching code. Because of virtualization, the original program is no longer code, but data interpreted by a virtual machine.

The delay incurred by the current implementation makes these techniques unusable for performance-critical parts of the program. There are however applications in DRM and license systems where this type of slowdown may be acceptable. These typically consist of some infrequently executed computations and one or more Boolean checks. The interest of industry for this type of technique and the usability of a production quality implementation has been proved by the recent acquisition of Secured Dimensions by Microsoft. The main product of Secured Dimensions is based on Virtual Machines.

6.2 Future Work

Since software diversity as a defense against malicious hosts is a relatively young domain, we expect many insights and discoveries to lay ahead. In this section, we take a look at possible extensions to the work presented in this dissertation to help further the domain.

An Evaluation Suite

Evaluation is one of the most challenging issues in software protection. A proof of a lower bound on the amount of work required by an attacker is very hard to obtain. Peer review may very well be the best available alternative. Furthermore, assessing the quality of a particular transformation merely from a description thereof is challenging. One really needs to be able to take a look at the resulting code for a real-life application.

Unfortunately, applications requiring software protection are often proprietary and closed source. Therefore, we advocate the development of an evaluation suite. This suite could include a number of different types of programs: a time-limited evaluation copy, a digital container with limited access to the content, a document processor that allows the viewing but not the printing of certain documents, a game, and so on.

We suggest to clearly define when these programs are considered to be broken. For example, the program is broken if it continues to work even if

the system clock is set after a particular date. This way it can be verified automatically whether or not an attack is successful.

At the time of writing, websites based upon similar ideas can be found on the web, e.g., http://www.crackmes.de. We suggest to extend these ideas and to promote them as a peer review channel for research.

Increasing the Number of False Positives

The goal of the transformations discussed in this dissertation is to make it harder to detect that two code fragments are related by transforming them to make sure that they are no longer perceived as identical. The primary goal is thus to increase the false negatives.

We have not discussed intentional attempts to make unrelated code fragments appear related. This could be an interesting research direction as well.

One suggested approach is to use a smaller instruction set. By using a smaller instruction set, we can increase the size of equivalence classes, a known cause for false positives. An instruction set such as the IA-32 contains a lot of redundant instructions. The instructions `call`, `ret`, `push`, `pop`, `inc`, `dec`, `sub`, ... can all be replaced by other instructions or instruction sequences. As a result, e.g., `call` instructions are no longer easily separated from other instructions, increasing the candidates with which they can be confused.

Generating a Small Number of Versions

In our experiments, the nonce generation was guided by a pseudorandom number generator. As a result, we are confident that we can generate many different versions for which the diversity between each pair is comparable. However, in some situations, a limited number of versions suffices. In the case of hiding the operation of a patch, for example, there are only two versions which should be as different as possible.

If we only need a limited number of versions, it would be beneficial to steer the diversity system to generate the most different versions. For example, instead of factoring functions with probability $p = 0.5$, we could keep track of which functions we have factored in one version and factor the other candidates in the other version.

Divide and Conquer Matching

The matching system discussed above considers the entire program at a time. It searches for related code fragments within the entire program. A more efficient strategy might be to split the program into smaller parts. For example, only the

code executed between two system or library calls of both versions can be considered. Code fragments from these portions of the code are then mapped onto each other without considering the rest of the code.

Matching Input Divergences

System calls proved to be a very reliable way to match code fragments. Unfortunately, they are not numerous. Another reliable source of information for matching may be the points where the execution diverges for different inputs. Sooner or later, the behavior of the program needs to depend on the input and different inputs may follow different execution paths. We believe that the points where the execution diverges because of difference in the input are likely to remain in diversified copies.

Canonicalization Attack

The matching system operates directly on the code of the versions. It may be improved by transforming both versions into some sort of canonical form before the matching. This canonical form may help to eliminate the impact of certain transformations such as instruction selection and scheduling. For example, we could transform the instructions into a small set of micro-operations. This could help, o.a., to identify the similarity between an increment and an add instruction with immediate one. Similarly, we could define a canonical way to schedule these micro-operations.

Targeted Attacks

We have deliberately kept the matching system generic and designed it without consideration of the specific diversifying transformations. If we know in advance what transformations will be applied, we can take this into account to improve the matching.

For example, if we know that control flow flattening has been applied, we can keep track of the different paths through the redirect block. We can then forward the edges and skip the redirect block to overcome the impact of control flow flattening on our matching system.

Bibliography

[Aho 86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Altinkemer 03] Kemal Altinkemer and Junwei Guan. Analyzing protection strategies for online software distribution. *Journal of Electronic Commerce Research*, 4(1):34–48, 2003.

[Anckaert 04a] Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Software piracy prevention through diversity. In *Proceedings of the 4th ACM Workshop on Digital Rights Management*, pages 63–71. ACM Press, 2004.

[Anckaert 04b] Bertrand Anckaert, Frederik Vandeputte, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Link-time optimization of IA64 binaries. In *Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 284–291. Springer-Verlag, 2004.

[Anckaert 05] Bertrand Anckaert, Bjorn De Sutter, Dominique Chanet, and Koen De Bosschere. Steganography for executables and code transformation signatures. In *Proceedings of the 7th International Conference on Information Security and Cryptology*, volume 3506 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, 2005.

[Anckaert 06] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the 6th ACM workshop on Digital Rights Management*, pages 47–58. ACM Press, 2006.

[Anckaert 07a] Bertrand Anckaert, Mariusz Jakubowski, Ramarathnam Venkatesan, and Koen De Bosschere. Run-time randomization to mitigate tampering. In *Proceedings of the 2nd International Workshop on Security*, volume 4752 of *Lecture Notes in Computer Science*, pages 153–168. Springer-Verlag, 2007.

[Anckaert 07b] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In *Proceedings of the 8th Information Hiding Conference*, volume 4437 of *Lecture Notes in Computer Science*, pages 232–248. Springer-Verlag, 2007.

[Anckaert 07c] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: A quantitative approach. In *Proceedings of the 3rd Workshop on Quality of Protection*, pages 15–20. ACM Press, 2007.

[Anderson 96] Ross Anderson and Markus Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the 2nd Usenix Workshop on Electronic Commerce*, pages 1–11, 1996.

[Anderson 98] Ross Anderson and Fabien Petitcolas. On the limits of steganography. *IEEE Journal of Selected Areas in Communications*, 16(4):474–481, 1998.

[Badger 01] Lee Badger, Larry D'Anna, Doug Kilpatrick, Brian Matt, Andrew Reisse, and Tom Van Vleck. Self-protecting mobile agents obfuscation evaluation report, 2001.

[Barak 01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st IACR Crypto Conference*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18, 2001.

[Barrantes 05] Elena Gabriela Barrantes, David Ackley, Stephanie Forrest, and Darko Stefanovi. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[Bhansali 06] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154–163. ACM Press, 2006.

[BSA 05] BSA (Business Software Alliance) and IDC (International Data Corporation). *Second Annual BSA and IDC Global Software Piracy Study*, 2005.

[Chang 02] Hoi Chang and Mikhail Atallah. Protecting software code by guards. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer-Verlag, 2002.

[Chen 02] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. In *Proceedings of the 5th Information Hiding Conference*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414. Springer-Verlag, 2002.

[Chow 01] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of the 4th Information Security Conference*, volume 2200 of *Lecture Notes in Computer Science*, pages 144– 155. Springer-Verlag, 2001.

[Chow 03] Stanley Chow, Philip Eisen, Harold Johnson, and Paul Van Oorschot. White-box cryptography and an AES implementation. In *Proceedings of the 9th Workshop on Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer-Verlag, 2003.

[Cifuentes 95] Cristina Cifuentes and John Gough. Decompilation of binary programs. *Software - Practice & Experience*, 25(7):811–829, 1995.

[Cohen 93] Frederick Cohen. Operating system evolution through program evolution. *Computers and Security*, 12(6):565–584, 1993.

[Collberg 98a] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 6th International Conference on Computer Languages*, pages 28–38. IEEE Computer Society Press, 1998.

[Collberg 98b] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th Conference on Principles of Programming Languages*, pages 184–196. ACM Press, 1998.

[Collberg 99] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Proceedings of the 26th Conference on Principles of Programming Languages*, pages 311–324. ACM Press, 1999.

[Collberg 03] Christian Collberg, Ginger Myles, and Andrew Huntwork. Sandmark - a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, 2003.

[Conner 91] Kathleen Conner and Richard Rumelt. Software piracy: an analysis of protection strategies. *Management Science*, 37(2):125–139, 1991.

[Cousot 03] Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In *Proceedings of the 30th Conference on Principles of Programming Languages*, pages 311–324. ACM Press, 2003.

[DiMarzio 07] Jerome DiMarzio. *The Debugger's Handbook*. Auerbach Publications, 2007.

[Dullien 05] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, page count 13, 2005.

[Dux 05] Bradley Dux, Anand Iyer, Saumya Debray, David Forrester, and Stephen Kobourov. Visualizing the behavior of dynamically modifiable code. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 337–340. IEEE Computer Society Press, 2005.

[El-Khalil 04] Rakan El-Khalil and Angelos Keromytis. Hydan: Information hiding in program binaries. In *Proceedings of the 6th International Conference on Informaton and Communications Security*, volume 3269 of *Lecture Notes in Computer Sciece*, pages 187–199. Springer-Verlag, 2004.

[Ernst 03] Michael Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of the 1st ICSE Workshop on Dynamic Analysis*, pages 25–28, 2003.

[Felten 03] Edward Felten. Understanding trusted computing: will its benefits outweigh its drawbacks. *IEEE Security and Privacy*, 1(03):60–62, 2003.

[Geer 03] Daniel Geer, R. Bace, Peter Gutmann, Perry Metzger, Charles Pfleeger, John Quarterman, and Bruce Schneier. Cyber insecurity: The cost of monopoly. Technical report, Computer & Communications Industry Association, 2003.

[Goldreich 96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[Graham 82] Susan Graham, Peter Kessler, and Marshall Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 2nd SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM Press, 1982.

[Heffner 04] Kelly Heffner and Christian Collberg. The obfuscation executive. In *Proceedings of the 7th Information Security Conference*, volume 3225 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 2004.

[Horne 02] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer-Verlag, 2002.

[IPR 03] IPR (International Planning and Research Corporation). *Software Management Guide*, 2003.

[Jakobsson 02] Markus Jakobsson and Michael Reiter. Discouraging software piracy using software aging. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, volume 2320 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2002.

[Katzenbeisser 00] Stefan Katzenbeisser and Fabien Petitcolas. *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.

[Kent 81] Stephen Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[Kerkhoffs 83] Auguste Kerkhoffs. La cryptographie militaire. *Journal de Sciences Militaires*, 9:5–38, 1883.

[Kruegel 04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of 13th USENIX Security Symposium*, pages 255–270, 2004.

[Leprosy 90] Leprosy. The Leprosy-B virus, 1990. http://familycode.atspace.com/lep.txt.

[Linn 03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 290–299. ACM Press, 2003.

[Madou 05a] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *Proceedings of the 6th International Workshop on Information Security Applications*, volume 3786 of *Lecture Notes in Computer Science*, pages 194–206. Springer-Verlag, 2005.

[Madou 05b] Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital Rights Management*, pages 75–82. ACM Press, 2005.

[Maebe 02] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Compendium of Workshops and Tutorials in Conjunction with the 11th International Conference on Parallel Architectures and Compilation Techniques*, page count 11, 2002.

[Massalin 87] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 122–126. IEEE Computer Society Press, 1987.

[Mic 02] Microsoft. *Microsoft Knowledge Base Article - 326904*, 2002. http://support.microsoft.com/kb/326904.

[Miller 85] Webb Miller and Eugene Myers. A file comparison program. *Software - Practice & Experience*, 15(11):1025–1040, 1985.

[Muchnick 97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Nagarajan 07] Vijayanand Nagarajan, Xiangyu Zhang, Rajiv Gupta, Matias Madou, Bjorn De Sutter, and Koen De Bosschere. Matching control flow of program versions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, pages 83–94. IEEE Computer Society Press, 2007.

[O'Donnell 04] Adam O'Donnell and Harish Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 121–131. ACM Press, 2004.

[Park 04] Yong-Joon Park and Gyungho Lee. Repairing return address stack for buffer overflow protection. In *Proceedings of the 1st ACM International Conference on Computing Frontiers*, pages 335–342. ACM Press, 2004.

[Patterson 90] David Patterson and John Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[Pettis 90] Karl Pettis and Robert Hansen. Profile guided code positioning. In *Proceedings of the 8th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27. ACM Press, 1990.

[Proebsting 95] Todd Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd Conference on Principles of Programming Languages*, pages 322–332. ACM Press, 1995.

[Ronsse 99] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions Computer Systems*, 17(2):133–152, 1999.

[Sabin 04] Todd Sabin. Comparing binaries with graph isomorphisms. Technical report, BindView RAZOR Team, 2004.

[Shacham 04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.

[SII 00] SIIA (Software and Information Industry Association). *Report on global software piracy*, 2000. http://www.siaa.net/.

[Simmons 84] Gustavus Simmons. The prisoners' problem and the subliminal channel. In *Advances in Cryptology, Proceedings of CRYPTO '83*, pages 51–67. Plenum Press, 1984.

[Sleator 85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[Sovarel 05] Ana Sovarel, David Evans, and Nathanael Paul. Where is the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, pages 145–160, 2005.

[van Oorschot 03] Paul van Oorschot. Revisiting software protection. In *Proceedings of the 6th Conference on Information Security*, volume 2851 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2003.

[Varadarajan 06] Avinash Varadarajan and Ramarathnam Venkatesan. Limited obliviousness for data structures and efficient execution of programs. Technical report, Microsoft Research, 2006.

[Venkatesan 01] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th Information Hiding Conference*, volume 2137 of *Lecture Notes in Computer Science*, pages 157–168. Springer-Verlag, 2001.

[Wagner 74] Robert Wagner and Michael Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[Wang 00] Zheng Wang, Ken Pierce, and Scott McFarling. Bmat – a binary matching tools for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[Wang 01] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Proceedings of the 2nd International Conference of Dependable Systems and Networks*, pages 193–202. IEEE Computer Society Press, 2001.

[Zeller 05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.

[Zhang 05] Xiangyu Zhang and Rajiv Gupta. Whole execution traces and their applications. *ACM Transactions on Architecture and Code Optimization*, 2(3):301–334, 2005.

[Zhou 06] Yongxin Zhou and Alec Main. Diversity via code transformations: A solution for NGNA renewable security. In *NCTA - The National Show*, 2006.